Bitcon Explanation:

```
def load_and_clean_data(file_path):
    """
    Loads data from all sheets in an Excel file and performs thorough
cleaning.
    Cleaning includes:
      - Converting 'Date' to datetime and dropping invalid dates.
      - Converting 'Price' to numeric and dropping invalid prices.
      - Aggregating duplicate dates within each sheet (using median).
      - Combining all sheets and aggregating duplicates across sheets.
      - Reindexing to a full daily date range and interpolating missing
values.
      - Removing outliers using the IQR method.
    """
    print("=== Loading and Cleaning Data ===")
    try:
        xls = pd.ExcelFile(file_path)
    except Exception as e:
        raise ValueError(f"Failed to open file '{file_path}': {e}")

    print("Available sheets in the Excel file:", xls.sheet_names)
    cleaned_dfs = []  # list to store cleaned DataFrames from each sheet

    for sheet in xls.sheet_names:
        print(f"\n--- Processing Sheet: '{sheet}' ---")
        try:
            df_sheet = pd.read_excel(file_path, sheet_name=sheet)
        except Exception as e:
            print(f"Error loading sheet '{sheet}': {e}")
            continue

        # Ensure required columns exist
        if 'Date' not in df_sheet.columns or 'Price' not in
df_sheet.columns:
            print(f"Sheet '{sheet}' skipped: Missing required columns
('Date' and/or 'Price').")
            continue

        # Convert 'Date' to datetime; drop rows with invalid dates
        df_sheet['Date'] = pd.to_datetime(df_sheet['Date'],
errors='coerce')
        num_invalid_dates = df_sheet['Date'].isna().sum()
        if num_invalid_dates > 0:
```

```
            print(f"Sheet '{sheet}': Dropping {num_invalid_dates} rows 
with invalid dates.")
        df_sheet.dropna(subset=['Date'], inplace=True)

        # Convert 'Price' to numeric; drop rows with invalid prices
        df_sheet['Price'] = pd.to_numeric(df_sheet['Price'], 
errors='coerce')
        num_invalid_prices = df_sheet['Price'].isna().sum()
        if num_invalid_prices > 0:
            print(f"Sheet '{sheet}': Dropping {num_invalid_prices} rows 
with invalid or missing Price values.")
        df_sheet.dropna(subset=['Price'], inplace=True)

        # Sort by date and set 'Date' as index
        df_sheet.sort_values('Date', inplace=True)
        df_sheet.set_index('Date', inplace=True)

        # Remove duplicate dates within this sheet using median (robust to 
outliers)
        initial_rows = df_sheet.shape[0]
        df_sheet = df_sheet.groupby(df_sheet.index).agg({'Price': 
'median'})
        final_rows = df_sheet.shape[0]
        if final_rows < initial_rows:
            print(f"Sheet '{sheet}': Aggregated duplicates within sheet 
(rows: {initial_rows} -> {final_rows}).")
        else:
            print(f"Sheet '{sheet}': No duplicate dates found.")
        print("Cleaned data preview:")
        print(df_sheet.head())
        cleaned_dfs.append(df_sheet)

    if not cleaned_dfs:
        raise ValueError("No valid sheets found after cleaning.")

    # Combine all sheets into one DataFrame
    combined_df = pd.concat(cleaned_dfs)
    print("\n--- Combined Data (Before Final Cleaning) ---")
    print(f"Combined data shape: {combined_df.shape}")
    print(combined_df.head())

    # Aggregate duplicates across sheets (again using median)
    initial_rows = combined_df.shape[0]
    combined_df = combined_df.groupby(combined_df.index).agg({'Price': 
'median'})
```

```
    final_rows = combined_df.shape[0]
    if final_rows < initial_rows:
        print(f"\nCombined data: Aggregated duplicates across sheets
(rows: {initial_rows} -> {final_rows}).")
    else:
        print("\nCombined data: No duplicate dates across sheets.")

    # Reindex to a full daily date range between the minimum and maximum
dates
    full_range = pd.date_range(start=combined_df.index.min(),
end=combined_df.index.max(), freq='D')
    combined_df = combined_df.reindex(full_range)
    print("\nAfter reindexing, data shape with full date range:",
combined_df.shape)

    # Interpolate missing values (using time interpolation)
    missing_before = combined_df['Price'].isna().sum()
    combined_df['Price'] = combined_df['Price'].interpolate(method='time')
    missing_after = combined_df['Price'].isna().sum()
    print(f"Filled missing values: {missing_before} -> {missing_after}
missing values.")

    # Remove outliers using the IQR method
    Q1 = combined_df['Price'].quantile(0.25)
    Q3 = combined_df['Price'].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    initial_rows = combined_df.shape[0]
    combined_df = combined_df[(combined_df['Price'] >= lower_bound) &
(combined_df['Price'] <= upper_bound)]
    final_rows = combined_df.shape[0]
    print(f"Outlier removal using IQR: Rows reduced from {initial_rows} to
{final_rows}.")

    print("\n=== Final Cleaned Data Summary ===")
    print(f"Total unique dates: {combined_df.shape[0]}")
    print("Date Range:", combined_df.index.min(), "to",
combined_df.index.max())
    print("Cleaned Data Preview:")
    print(combined_df.head())

    return combined_df

def create_sequences(dataset, seq_length):
```

```
    """
    Creates sequences of length `seq_length` for time series forecasting.
    Returns arrays for input features (X) and target values (y).
    """
    X, y = [], []
    for i in range(seq_length, len(dataset)):
        X.append(dataset[i - seq_length:i, 0])
        y.append(dataset[i, 0])
    return np.array(X), np.array(y)

def fetch_current_bitcoin_price():
    """
    Fetches the current Bitcoin price in USD using the Coingecko API.
    """
    url =
"https://api.coingecko.com/api/v3/simple/price?ids=bitcoin&vs_currencies=u
sd"
    try:
        response = requests.get(url, timeout=10)
        response.raise_for_status()
        data = response.json()
        return data["bitcoin"]["usd"]
    except Exception as e:
        print("Error fetching real-time data:", e)
        return None

# Global sequence length variable (used later in model building)
seq_length = 60

def build_model(hp):
    """
    Build and compile the LSTM model. This function is used by KerasTuner.
    """
    model = Sequential()
    # Tune the number of LSTM units and dropout rate
    lstm_units = hp.Int('lstm_units', min_value=32, max_value=128,
step=32, default=64)
    dropout_rate = hp.Float('dropout_rate', min_value=0.1, max_value=0.5,
step=0.1, default=0.3)

    model.add(LSTM(units=lstm_units, return_sequences=True,
input_shape=(seq_length, 1)))
    model.add(Dropout(dropout_rate))
    model.add(LSTM(units=lstm_units, return_sequences=False))
    model.add(Dropout(dropout_rate))
```

```
    # Tune the dense layer size
    dense_units = hp.Int('dense_units', min_value=16, max_value=64,
step=16, default=32)
    model.add(Dense(units=dense_units, activation='relu'))
    model.add(Dense(units=1))

    # Tune the learning rate
    learning_rate = hp.Float('learning_rate', min_value=1e-4,
max_value=1e-2, sampling='log', default=1e-3)
    model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=learnin
g_rate),
                  loss='mean_squared_error')
    return model
```

1. Loading and Cleaning Data

Function: load_and_clean_data(file_path) This function loads data from an Excel file that contains multiple sheets, each representing a different dataset. The function performs several cleaning steps:

- **Loading Data:** It reads all sheets from the Excel file and prints the sheet names.
- **Filtering Necessary Columns:** Ensures each sheet has 'Date' and 'Price' columns, skipping sheets that do not.
- **Date & Price Conversion:** Converts the 'Date' column to a datetime format and 'Price' to numeric. Rows with invalid data are removed.
- **Sorting & Indexing:** Sorts the data by date and sets 'Date' as the index.
- **Removing Duplicates:** If there are multiple entries for the same date, it takes the median value to handle outliers.
- **Merging Data:** Combines cleaned data from all sheets.
- **Handling Missing Dates:** Reindexes to include all missing dates and fills them using time-based interpolation.

- **Outlier Removal:** Uses the **Interquartile Range (IQR)** method to remove extreme values.

We need a clean and structured dataset for accurate time-series analysis and forecasting. The dataset is processed to remove inconsistencies, handle missing values, and normalize prices.

2. Sequence Preparation for Forecasting

Function: create_sequences(dataset, seq_length)

This function prepares the dataset for time-series forecasting using sequences of past prices to predict future prices.

LSTM models require sequential data as input. This function converts historical price data into feature-target pairs for training.

3. Fetching Real-Time Bitcoin Price

Function: fetch_current_bitcoin_price()

This function calls the CoinGecko API to fetch the current Bitcoin price in USD.

Why use an API? Real-time data is crucial for predicting future trends. Instead of relying only on historical data, this function allows integration of the latest market price.

4. LSTM Model for Forecasting

Function: build_model(hp)

This function builds a Long Short-Term Memory (LSTM) model with tunable hyperparameters for price prediction.

- **LSTM Layers:** Extract patterns from historical data.
- **Dropout Layers:** Prevents overfitting.
- **Dense Layers:** Help in final prediction.
- **Adam Optimizer:** Ensures efficient training with an optimal learning rate.

Why LSTM? LSTMs are effective in time-series forecasting because they capture long-term dependencies in data.