PROJECT – ETL PIPELINE WITH PYTHON

Data integration is the process of combining data from different sources into a unified, cohesive view that can be accessed and analysed across an organization. This process ensures that information is **consistent**, **accurate** and **readily available** (for decision-making, reporting, and analytics)

Data integration is relevant when data is dispersed across multiple platforms and departments. It helps to break down silos and improve operational efficiency, which results in smoother communication between systems and teams. And with a comprehensive view of their data, people are better positioned to make informed, strategic decisions.

ETL is the traditional data integration technique. It involves three key steps:

- Extracting data from various sources
- Transforming the data into a common format
- Loading the data into a target storage, normally a data warehouse

In the extraction phase, data is collected from different systems. The transformation phase ensures the data is cleaned, standardized, and formatted to fit a consistent schema. Finally, in the load phase, the transformed data is stored in the target system, where it becomes available for analysis and reporting.

This project uses the Alpha Vantage Intraday Time Series API. This API returns current and 20+ years of historical intraday OHLCV time series of the ticker of the equity specified, in this case IBM, covering pre-market and post-market hours where applicable (e.g., 4:00am to 8:00pm Eastern Time for the US market).

Steps taken:

The first step imports the necessary libraries. `requests` is used to make API calls, `pandas` handles data manipulation, and `sqlalchemy` enables interaction with the SQLite database.

```
# Import required libraries for API access, data manipulation, and database interaction
import requests
import pandas as pd
from sqlalchemy import create_engine, text
```

This step defines key parameters needed to request data from the Alpha Vantage API. The `url` is dynamically constructed using the base endpoint and query parameters like symbol, function type, interval, and API key.

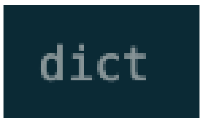
```python
API_KEY = "YOUR_API_KEY"  # Replace with your actual API key
# API configuration
SYMBOL = "IBM"
FUNCTION = "TIME_SERIES_INTRADAY"
INTERVAL = "1min"
BASE_URL = "https://www.alphavantage.co/query"

url = f"{BASE_URL}?function={FUNCTION}&symbol={SYMBOL}&interval={INTERVAL}&apikey={API_KEY}"
```

This code sends a GET request to the Alpha Vantage API using the constructed URL. It checks the HTTP status code to confirm the request was successful. If so, the JSON response is stored in `api_data_1`. Otherwise, it prints an error with the status code.

```python
# API request
response = requests.get(url)

# Check if the request was successful
if response.status_code == 200:
    api_data_1 = response.json()
else:
    print(f"Error: Unable to fetch data. HTTP Status Code: {response.status_code}")

response.status_code
```

This line checks the type of the `api_data_1` object using Python's built-in `type()` function. It's used here to confirm that the API response was successfully parsed into a Python dictionary, which is the expected format for further data extraction.

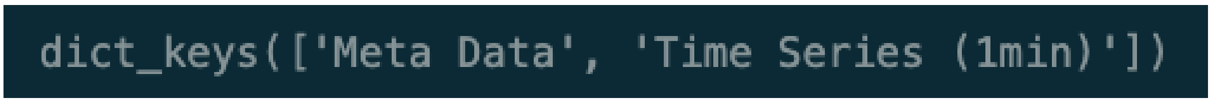
```python
# Check the type of api_data_1
type(api_data_1)
```



```
dict
```

After confirming that `api_data_1` is a dictionary, this line lists its top-level keys. This helps understand the structure of the API response and locate the specific section (like the time series data) needed for further processing.

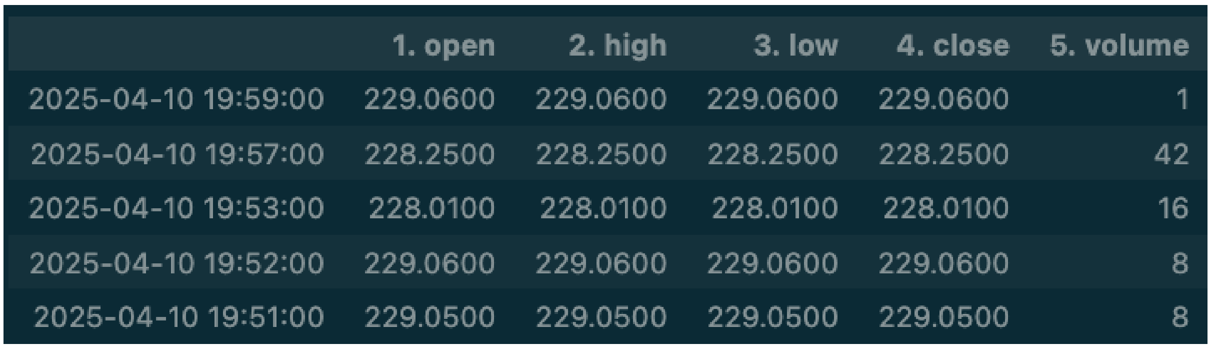
```python
api_data_1.keys()
```



```
dict_keys(['Meta Data', 'Time Series (1min)'])
```

The time series data is stored under the `"Time Series (5min)"` key in the API response. This line extracts that nested dictionary and converts it into a transposed `DataFrame`, so each row represents one timestamped entry. The `.head()` call displays the first few rows to verify the structure.

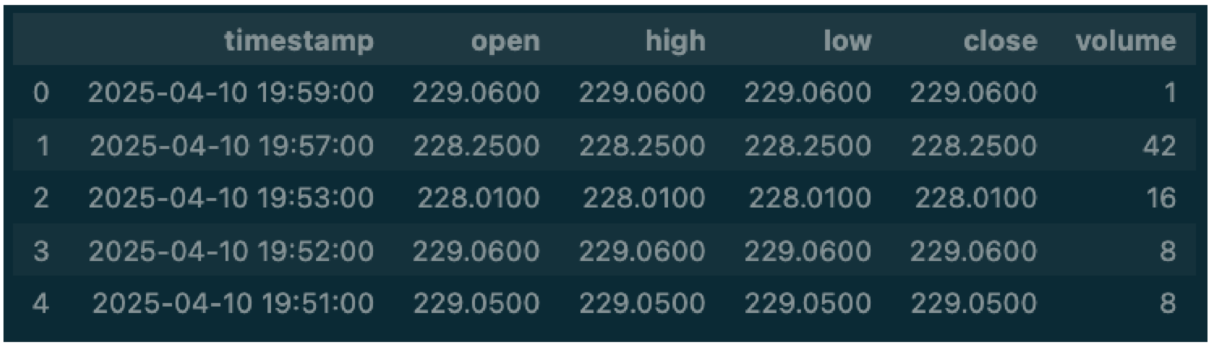
```
# Extracting the time series data
df = pd.DataFrame(api_data_1["Time Series (1min)"]).T
df.head()
```



	1. open	2. high	3. low	4. close	5. volume
2025-04-10 19:59:00	229.0600	229.0600	229.0600	229.0600	1
2025-04-10 19:57:00	228.2500	228.2500	228.2500	228.2500	42
2025-04-10 19:53:00	228.0100	228.0100	228.0100	228.0100	16
2025-04-10 19:52:00	229.0600	229.0600	229.0600	229.0600	8
2025-04-10 19:51:00	229.0500	229.0500	229.0500	229.0500	8

This block cleans and prepares the `DataFrame` for further processing. Column names are simplified for readability, the datetime index is reset and renamed to `"timestamp"`, and it is explicitly converted to a datetime object. This ensures compatibility with SQL and makes time-based operations easier.

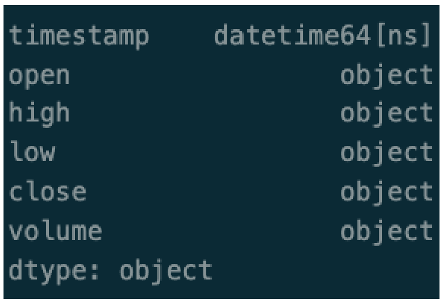
```
# rename columns
df.columns = ["open", "high", "low", "close", "volume"]
# reset index
df.reset_index(inplace=True)
# rename index column
df.rename(columns={"index": "timestamp"}, inplace=True)
# convert timestamp to datetime
df["timestamp"] = pd.to_datetime(df["timestamp"])

df.head()
```



	timestamp	open	high	low	close	volume
0	2025-04-10 19:59:00	229.0600	229.0600	229.0600	229.0600	1
1	2025-04-10 19:57:00	228.2500	228.2500	228.2500	228.2500	42
2	2025-04-10 19:53:00	228.0100	228.0100	228.0100	228.0100	16
3	2025-04-10 19:52:00	229.0600	229.0600	229.0600	229.0600	8
4	2025-04-10 19:51:00	229.0500	229.0500	229.0500	229.0500	8

This line displays the current data types of each column in the `DataFrame`. It helps verify that numeric fields like `open`, `high`, `low`, etc., are not mistakenly stored as objects (strings), and confirms that `timestamp` is properly recognized as a datetime type.

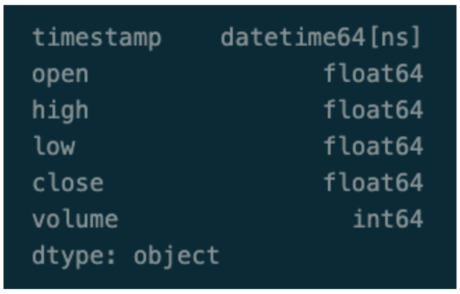
```
df.dtypes
```



```
timestamp    datetime64[ns]
open                 object
high                 object
low                  object
close                object
volume               object
dtype: object
```

This step ensures that all numerical columns are explicitly converted to numeric types (`float64` or `int64`), which is essential for compatibility with database storage. Using `errors="coerce"` handles any invalid entries gracefully by converting them to `NaN`, preventing insertion errors during the loading process. The final `df.dtypes` confirms that the conversion was successful.

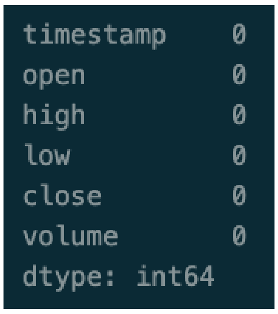
```
df[["open", "high", "low", "close", "volume"]] = df[["open", "high", "low",
"close", "volume"]].apply(pd.to_numeric, errors="coerce")
df.dtypes
```



```
timestamp    datetime64[ns]
open                float64
high                float64
low                 float64
close               float64
volume                int64
dtype: object
```

This step checks for missing values in the `DataFrame`. By chaining `.isnull()` with `.sum()`, it returns the count of `NaN` values in each column. This helps identify any data quality issues before inserting the data into the database.

```
df.isnull().sum()
```



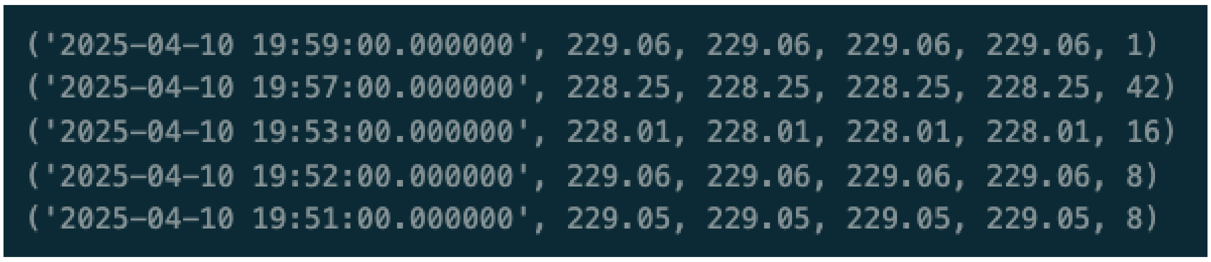
```
timestamp    0
open         0
high         0
low          0
close        0
volume       0
dtype: int64
```

This block creates a connection to a SQLite database using SQLAlchemy and writes the cleaned DataFrame to a table named based on the stock symbol (e.g., `"time_series_intraday_ibm"`). The database file is also named dynamically using the symbol, which helps keep datasets organized and separate for each stock. The `if_exists="replace"` parameter ensures any existing table with the same name is overwritten, and `index=False` prevents the default row index from being written to the database, since the timestamp column is already present.

```
# Create a SQLite database engine
engine = create_engine(f"sqlite:///time_series_intraday_{SYMBOL.lower()}.db",
future=True)
# Load the DataFrame into the database
df.to_sql(f"time_series_intraday_{SYMBOL.lower()}", con=engine,
if_exists="replace", index=False)
```

To confirm that the data was successfully loaded into the database, a simple `SELECT` query was executed using SQLAlchemy. This retrieves a few rows from the newly created table and validates that the data is correctly structured and accessible.

```
# Preview a few rows from the final database table
with engine.connect() as conn:
    result = conn.execute(text(f"SELECT * FROM
time_series_intraday_{SYMBOL.lower()} LIMIT 5"))
    for row in result:
        print(row)
```



```
('2025-04-10 19:59:00.000000', 229.06, 229.06, 229.06, 229.06, 1)
('2025-04-10 19:57:00.000000', 228.25, 228.25, 228.25, 228.25, 42)
('2025-04-10 19:53:00.000000', 228.01, 228.01, 228.01, 228.01, 16)
('2025-04-10 19:52:00.000000', 229.06, 229.06, 229.06, 229.06, 8)
('2025-04-10 19:51:00.000000', 229.05, 229.05, 229.05, 229.05, 8)
```

With the data cleaned, transformed and written to a database, the core ETL pipeline is complete and ready for validation or downstream use.dp

Core lessons from this project:

- Hands-on with end-to-end API workflows: Learned how to construct, send and validate API requests, and how to interpret and extract nested JSON responses.
- Data cleaning and schema alignment: Understood the importance of explicitly setting data types before inserting into a database to avoid data mismatches
- `SQLAlchemy` and `.to_sql()` mechanics: Learned how `SQLAlchemy` integrates with pandas and how parameters like `if_exists` and `index` affect the output schema.
- Workflow troubleshooting: Learned how to diagnose issues like unexpected column types and format mismatches, and fixing them

Potential Improvements & Optimizations:

- Environment variable for API key: Instead of hardcoding the API key, storing it securely using environment variables or a `config` file would be beneficial
- API rate limit: For any API there are request limits in place. While the current script doesn't account for this, implementing a delay or backoff strategy would help avoid hitting the limit.
- Command-line arguments: The current script is hardcoded to pull data for a specific symbol. Making the symbol more dynamic via command-line `args` or a simple GUI would make it more user-friendly.

- Support for multiple symbols: Extending the script to loop over a list of symbols/tickers and building separating tables in the same DB or separate DBs altogether would make the project scalable

With the data successfully loaded into a structured database, this pipeline is now ready to support further analysis, reporting, or integration with downstream applications.