Stock Market Time Series Analysis with Pandas

Notebook 01: Data Loading, Inspection, and Quality Check

Part of the comprehensive learning series: Stock Market Time Series Analysis with Pandas

Learning Objectives:

- Master data fetching with `yfinance`
- Handle MultiIndex columns from financial data sources
- Perform initial data quality checks and cleaning
- Create foundational visualizations for trend analysis

- Welcome to the first notebook in our time series analysis journey! The foundation of any successful data project is **good data**, and for finance, that means real, clean, and reliable historical prices.
- In this notebook, we will cover the essential first steps:
    1. **Fetching Data:** Using the `yfinance` library to download historical stock prices.
    2. **Initial Inspection:** Using core Pandas methods (`.info()` , `.describe()`) to understand the data's structure and types, specifically addressing the **MultiIndex columns** often produced by `yfinance` .
    3. **Data Cleaning:** Flattening the column names for easier access.
    4. **Data Quality:** Checking for missing values (`NaN`) and spotting early trends/volatility through basic plots.

1. Setup and Data Fetching

- We need `yfinance` to grab the data and `pandas` for handling it.
- We'll use **Apple (AAPL)** for a wide time range.

In [1]:
```
# Import necessary libraries
import pandas as pd
import yfinance as yf
```

```
import matplotlib.pyplot as plt
import seaborn as sns
```

In [2]:
```
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
```

In [3]:
```
# Set a specific theme for cleaner visualizations
sns.set_style('whitegrid')

# --- Concept: Define the Data ---
TICKER = 'AAPL'
START_DATE = '2019-01-01'
END_DATE = '2025-01-01'

# --- Code: Fetching Data ---
print(f"Fetching historical data for {TICKER}...")
df = yf.download(
    TICKER,
    start=START_DATE,
    end=END_DATE,
    # Note: Setting interval='1d' is often default, but helpful for clarity
    interval='1d'
)
```

```
Fetching historical data for AAPL...
[*********************100%***********************]  1 of 1 completed
```

In [4]:
```
print("\nFirst 5 rows of the DataFrame:")
df.head()
```

```
First 5 rows of the DataFrame:
```

Out[4]:

Price	Close	High	Low	Open	Volume
Ticker	AAPL	AAPL	AAPL	AAPL	AAPL
Date					
2019-01-02	37.575211	37.796495	36.697218	36.854258	148158800
2019-01-03	33.832447	34.672369	33.787238	34.258355	365248800
2019-01-04	35.276722	35.345726	34.215519	34.389213	234428400
2019-01-07	35.198208	35.412354	34.715193	35.381421	219111200
2019-01-08	35.869190	36.123786	35.338589	35.586043	164101200

Insights from Fetched Data (Column Structure)

- Notice the column header structure. Because we fetched data for only one ticker (AAPL), `yfinance` has returned a DataFrame with a **MultiIndex on the columns**.
- The top level shows the metric (`Close` , `High` , `Low` , `Open` , `Volume`), and the second level shows the ticker (`AAPL`).

- In the typical `yfinance` output, we usually see six columns: `Open`, `High`, `Low`, `Close`, `Adj Close`, and `Volume`.
- This current dataset structure is slightly different, missing the explicit `Adj Close` column.
- **For this notebook,** we will use the `Close` price as the main price series for simplicity, recognizing it is unadjusted for dividends. (If the `Adj Close` were present, we would always use it for long-term analysis.)
- **Key Columns:** We have the crucial Open, High, Low, Close (price information) and Volume (liquidity information).

2. Data Cleaning: Flattening Column Names

- Accessing columns like `df[('Close', 'AAPL')]` is cumbersome.
- We must simplify the column headers to make the code cleaner and more readable.

In [5]:
```
# Let's inspect the column structure at level 0
df.columns.get_level_values(0)
```

Out[5]:
```
Index(['Close', 'High', 'Low', 'Open', 'Volume'], dtype='object', name='Pric
e')
```

In [6]:
```
# Let's inspect the column structure at level 1
df.columns.get_level_values(1)
```

Out[6]:
```
Index(['AAPL', 'AAPL', 'AAPL', 'AAPL', 'AAPL'], dtype='object', name='Ticker')
```

In [7]:
```
# --- Concept: Flattening MultiIndex Columns ---
# We'll take the top level of the MultiIndex (Close, High, Low, Open, Volume)
# and assign them as the new, single-level column names.

# --- Code: Rename Columns ---
df.columns = df.columns.get_level_values(0)

print("\nUpdated Column Headers:")
print(df.columns.tolist())

print("\nFirst 5 rows after cleaning columns:")
df.head()
```

```
Updated Column Headers:
['Close', 'High', 'Low', 'Open', 'Volume']

First 5 rows after cleaning columns:
```

Out[7]:

Price	Close	High	Low	Open	Volume
Date					
2019-01-02	37.575211	37.796495	36.697218	36.854258	148158800
2019-01-03	33.832447	34.672369	33.787238	34.258355	365248800
2019-01-04	35.276722	35.345726	34.215519	34.389213	234428400
2019-01-07	35.198208	35.412354	34.715193	35.381421	219111200
2019-01-08	35.869190	36.123786	35.338589	35.586043	164101200

3. Initial Data Inspection (The Pandas Way)

- Now that the column names are clean, we use `.info()` and `.describe()` to check structure and statistics.

In [8]:
```
# --- Code: Data Structure and Types (df.info()) ---
print("\n--- DataFrame Information ---")

df.info()
```

```
--- DataFrame Information ---
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1510 entries, 2019-01-02 to 2024-12-31
Data columns (total 5 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   Close   1510 non-null   float64
 1   High    1510 non-null   float64
 2   Low     1510 non-null   float64
 3   Open    1510 non-null   float64
 4   Volume  1510 non-null   int64  
dtypes: float64(4), int64(1)
memory usage: 70.8 KB
```

Insights from `df.info()`

1. **Index Type:** The index is confirmed as `DatetimeIndex`, which is the core of time series analysis in Pandas.

2. **Completeness:** The **Non-Null Count** is the same for all columns, suggesting no missing values within the trading days provided by the source. All data types are numeric (`float64` or `int64`).

In [9]:
```
# --- Code: Descriptive Statistics (df.describe()) ---
print("\n--- Descriptive Statistics ---")
# .T transposes the result for better readability
# Using .style.format to limit decimal places for cleaner output
df.describe().T.style.format("{:.2f}")
```

```
--- Descriptive Statistics ---
```

Out[9]:

Price	count	mean	std	min	25%	50%
Close	1510.00	134.82	54.05	33.83	88.66	143.09
High	1510.00	136.15	54.44	34.67	90.01	144.67
Low	1510.00	133.32	53.58	33.79	88.02	141.42
Open	1510.00	134.67	54.00	34.26	88.61	142.94
Volume	1510.00	94167751.26	52325542.56	23234700.00	59144200.00	81521050.00

Insights from `df.describe()`

1. **Scale Difference:** The `Volume` column has a mean in the hundreds of millions, while prices are in the hundreds. This massive scale difference confirms the need for separate axes or normalization in visualization.

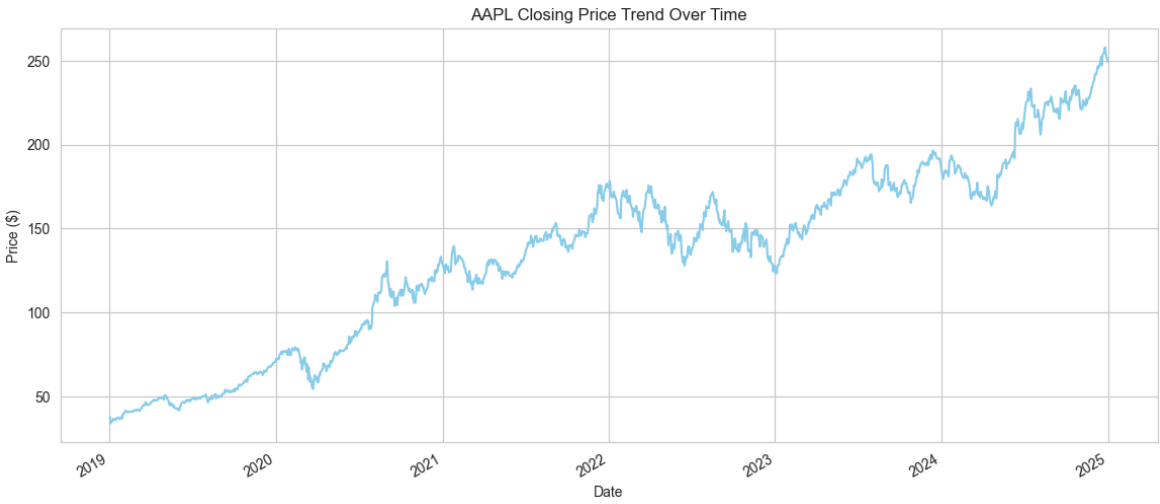
2. **Price Range:** The difference between the minimum and maximum price gives a raw sense of the growth and overall price movement over the 6-year period.

4. Basic Visualization for Trend and Volatility

- The simplest plots provide immediate insight into the overall trend and daily volatility.

In [10]:
```
# --- Code: Plot Raw Close Price ---

plt.figure(figsize=(14, 6))
# We plot the 'Close' price as our main series
df['Close'].plot(
    title=f'{TICKER} Closing Price Trend Over Time',
    color='skyblue',
    linewidth=1.5
)
plt.xlabel('Date')
plt.ylabel('Price ($)')
plt.show()
```



Visualization 1 Insights

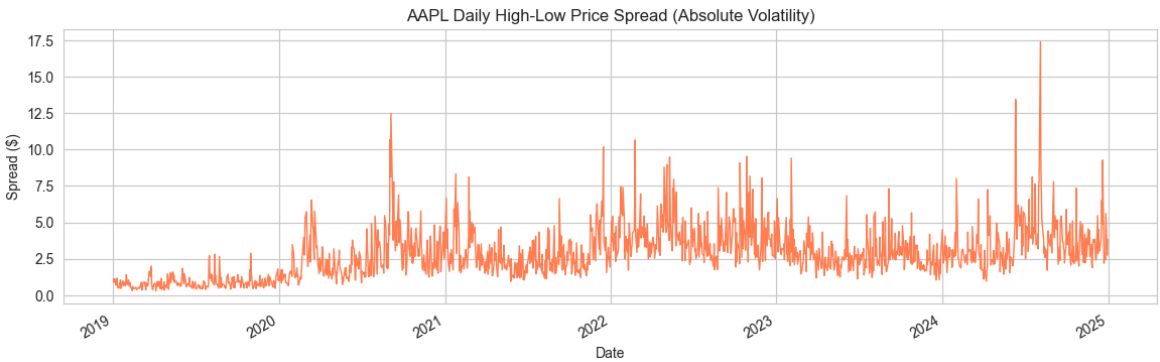
1. **Overall Trend:** The plot clearly shows a **long-term bullish (upward) trend**, a fundamental characteristic of successful growth stocks.
2. **Market Events:** Key drops, such as the initial 2020 crash, are clearly visible. The speed of recovery after such drops can be a good indicator of the stock's resilience and momentum.
3. **Visual Volatility:** The magnitude of day-to-day changes (volatility) seems to increase as the stock price itself rises, suggesting higher risk *in absolute dollar terms*.

In [11]:
```
# --- Code: Plot Daily High/Low Spread ---

# Calculate the difference between the High and Low price for each day (a proxy
# This gives us a simple measure of absolute volatility.

df['High_Low_Spread'] = df['High'] - df['Low']

plt.figure(figsize=(14, 4))
df['High_Low_Spread'].plot(
    title=f'{TICKER} Daily High-Low Price Spread (Absolute Volatility)',
    color='coral',
    linewidth=1
)
plt.xlabel('Date')
plt.ylabel('Spread ($)')
plt.show()
```



Visualization 2 Insights

1. **Volatility Spikes:** The highest spikes in the spread plot often correspond to periods of market instability or major news events (e.g., earnings releases).
2. **Volatility Trend:** The general *floor* of the daily spread has moved higher over time, confirming the observation from the main price chart: this stock requires a larger daily price movement in dollars to cover its range, which is a sign of increasing absolute volatility.

5. Summary and Next Steps

Key Takeaways

- **Data Preparation:** We successfully handled the **MultiIndex columns** from the data source by flattening them, a crucial Pandas skill for financial data.
- **Data Health:** The data is clean, indexed correctly, and ready for advanced time series operations.
- **Initial Findings:** The stock exhibits a strong upward trend and increasing absolute volatility.

Next Notebook Preview

- Now that we have clean data, the next step is to master the **Time Index** itself.
- We will dedicate the next notebook to **indexing, time slicing (filtering by date), and extracting time-based features** (year, month, weekday) using powerful Pandas features.

About This Project

This notebook is part of the **Stock Market Time Series Analysis with Pandas** repository - a comprehensive, beginner-to-intermediate friendly guide for mastering financial time series analysis using Python and Pandas.

Repository: `stock-time-series-analysis-with-pandas`

Author

Prakash Ukhalkar

GitHub prakash-ukhalkar
