**Q1. What is the distinction between a numpy array and a pandas data frame? Is there a way to convert between the two if there is?**

* **NumPy Array**:
  + A **NumPy array** is a powerful n-dimensional array object in Python that is mainly used for numerical computations. It can hold only one type of data (e.g., integers, floats, etc.) and is designed for efficient numerical operations.
  + It is best suited for vectorized operations and mathematical computations.
* **Pandas DataFrame**:
  + A **Pandas DataFrame** is a two-dimensional, size-mutable, and potentially heterogeneous tabular data structure. It can hold multiple types of data (e.g., integers, floats, strings) in different columns, and each column is a pandas.Series object.
  + It is more suitable for handling structured data like datasets that come from Excel, CSV files, or SQL databases.
* **Converting between NumPy array and Pandas DataFrame**:
  + You can convert a **NumPy array** to a **Pandas DataFrame** using pd.DataFrame().
  + import pandas as pd
  + import numpy as np
  + array = np.array([[1, 2], [3, 4], [5, 6]])
  + df = pd.DataFrame(array, columns=['A', 'B'])
  + You can convert a **Pandas DataFrame** to a **NumPy array** using .values or .to\_numpy().
  + df = pd.DataFrame([[1, 2], [3, 4], [5, 6]], columns=['A', 'B'])
  + array = df.to\_numpy()

**Q2. What can go wrong when a user enters a stock-ticker symbol, and how do you handle it?**

When a user enters a **stock-ticker symbol**, the following things might go wrong:

1. **Invalid symbol**: The symbol may not exist or be invalid, which could result in a 404 or symbol not found error.
   * **Handling**: You can check for valid stock symbols by using an API that validates them (like Yahoo Finance or Alpha Vantage). If invalid, raise an exception and prompt the user to try again.
2. **Incorrect format**: The symbol might be entered in the wrong case or with extra spaces.
   * **Handling**: You can sanitize the input by stripping spaces and converting the symbol to uppercase.
3. **Network issues**: The user may face connectivity issues when trying to fetch stock data.
   * **Handling**: You can use **try-except** blocks to catch network-related errors and give feedback to the user.

Example code:

import yfinance as yf

try:

ticker = "AAPL" # Example ticker

stock\_data = yf.Ticker(ticker).history(period="1d")

except ValueError:

print("Invalid stock ticker!")

except Exception as e:

print(f"An error occurred: {e}")

**Q3. Identify some of the plotting techniques that are used to produce a stock-market chart.**

Some common plotting techniques for creating stock-market charts include:

1. **Line Chart**: Used to visualize the stock's price over time. The X-axis represents time, and the Y-axis represents the stock price.
   * Typically used to show the trend of a stock's value over a period of time.
2. **Candlestick Chart**: A financial chart used to show the open, high, low, and close prices for a stock over a specific time period.
   * Used to analyze the price movement and predict future trends.
3. **Bar Chart**: Displays the opening, closing, high, and low prices for a stock during a specific time period, similar to the candlestick chart but using bars.
4. **Volume Chart**: Often placed below the price chart, showing the trading volume over time, which helps to understand the liquidity of the stock.
5. **Moving Average**: A plot that represents the average of stock prices over a certain period (e.g., 50-day or 200-day moving averages) to smooth out price fluctuations.

import yfinance as yf

import matplotlib.pyplot as plt

# Fetch stock data

ticker = "AAPL"

stock\_data = yf.Ticker(ticker).history(period="1mo")

# Plotting line chart for the 'Close' price

stock\_data['Close'].plot(title=f'{ticker} Stock Price')

plt.show()

**Q4. Why is it essential to print a legend on a stock-market chart?**

A **legend** is essential on a stock-market chart because it helps to **identify and differentiate** various data series or lines that might be plotted on the same graph. For example:

* If you plot both the **stock price** and the **moving average**, a legend will indicate which line corresponds to which data.
* It adds **clarity** and **context** to the chart, making it easier for users to understand the meaning of different lines and markers.

stock\_data['Close'].plot(label='Close Price')

stock\_data['50MA'] = stock\_data['Close'].rolling(window=50).mean()

stock\_data['50MA'].plot(label='50-day Moving Average')

plt.legend()

plt.show()

**Q5. What is the best way to limit the length of a pandas data frame to less than a year?**

You can limit the length of a **Pandas DataFrame** to less than a year by filtering the data based on **date**.

* You can use **datetime operations** to filter rows for the last year or a specific period.

Example:

import pandas as pd

# Sample DataFrame with dates

df = pd.DataFrame({

'date': pd.date\_range(start="2021-01-01", periods=500, freq='D'),

'data': range(500)

})

# Limit to the last 1 year (365 days)

df = df[df['date'] > pd.Timestamp.today() - pd.Timedelta(days=365)]

This will keep only the rows where the date is within the last 365 days.

**Q6. What is the definition of a 180-day moving average?**

A **180-day moving average** is the average of a stock's price over the past **180 days**, recalculated every day. It is commonly used to smooth out price fluctuations and identify longer-term trends. It is calculated by averaging the closing prices of the last 180 trading days.

Formula:

180-day MA=∑i=1180Pricei180\text{180-day MA} = \frac{\sum\_{i=1}^{180} \text{Price}\_{i}}{180}

**Q7. Did the chapter's final example use "indirect" importing? If so, how exactly do you do it?**

Yes, **indirect importing** is a technique where modules are imported indirectly via a dynamic or runtime approach, rather than through a static import statement. This is commonly done using the importlib module in Python.

Example:

import importlib

# Indirectly import a module

module\_name = "math"

math\_module = importlib.import\_module(module\_name)

# Use the imported module

result = math\_module.sqrt(16)

print(result) # Output: 4.0

In this example, the module name ("math") is provided as a string and the module is imported dynamically using importlib.import\_module(). This technique is useful when the module name is not known ahead of time.