1. What is web scraping. Describe the steps to perform web scraping.

**Web Scraping (10 Marks)**

**Introduction:** Web scraping is the process of automatically extracting data from websites. It allows users to gather information from web pages and store it in a structured format for analysis, such as in a database or a spreadsheet. Web scraping is widely used in applications like data analysis, price comparison, lead generation, and research. The data is often scraped using automated tools and scripts, typically written in languages like Python, which help navigate and extract the required information from HTML pages.

**Steps to Perform Web Scraping:**

1. **Identify the Target Website:**
   * Choose a website from which data needs to be extracted. Analyze the structure and format of the website, ensuring that the data is publicly available and that scraping the site complies with its terms of service.
2. **Inspect the Web Page:**
   * Use the browser’s developer tools (e.g., by right-clicking and selecting “Inspect”) to examine the HTML structure of the web page. Identify the specific tags (such as <div>, <table>, <span>, etc.) that contain the data you want to scrape.
   * For example, in an e-commerce site, product names and prices may be stored in specific HTML elements.
3. **Choose the Appropriate Tools or Libraries:**
   * Depending on the programming language, choose appropriate libraries or frameworks for web scraping. In Python, popular libraries include:
     + BeautifulSoup: For parsing HTML and extracting data.
     + Selenium: For simulating browser interaction and scraping dynamic content.
     + Scrapy: A comprehensive framework for large-scale scraping projects.
     + Requests: For sending HTTP requests to fetch web pages.
4. **Send HTTP Requests:**
   * Use tools like Requests to send HTTP GET requests to the server and retrieve the HTML content of the web page. The response from the server contains the raw HTML data.
   * **import requests**
   * **response = requests.get('https://example.com')**
   * **html\_content = response.content**

**Parse the HTML Content:**

* Once the HTML content is fetched, parse it using a library like BeautifulSoup. This allows for easy navigation of the document tree and extraction of data from specific tags and attributes.

**from bs4 import BeautifulSoup**

**soup = BeautifulSoup(html\_content, 'html.parser')**

**titles = soup.find\_all('h1') # Example for extracting all <h1> tags**

**Extract the Required Data:**

* Use the parsed HTML to locate and extract the relevant data. This might involve searching for specific tags, classes, or IDs that contain the information.

**prices = soup.find\_all('span', class\_='price') # Extracting product prices**

**Handle Pagination (if applicable):**

* Many websites split data across multiple pages. If the target website uses pagination, identify the URL patterns or navigation buttons and write logic to scrape all relevant pages iteratively.

**Store the Data:**

* After extraction, store the data in a structured format, such as a CSV file, JSON file, or database.
* Example:

**import csv**

**with open('data.csv', 'w') as file:**

**writer = csv.writer(file)**

**writer.writerow(['Title', 'Price'])**

**for title, price in zip(titles, prices):**

**writer.writerow([title.text, price.text])**

**Handle JavaScript Content (if applicable):**

* Some websites load content dynamically using JavaScript. Tools like Selenium can be used to simulate a browser environment and extract data after JavaScript execution.

**Respect Ethical and Legal Boundaries:**

* Always check the website’s robots.txt file to understand the permissions for crawling or scraping. Ensure compliance with legal regulations, including copyright laws and terms of service, and avoid overloading servers with excessive requests (use delays between requests).

1. Discuss the challenges in performing web scraping.

**Challenges in Performing Web Scraping (10 Marks)**

**Introduction:** Web scraping, while an effective method of extracting data from websites, comes with several challenges and limitations. These challenges can arise from legal, technical, or ethical considerations, making it crucial for developers to address them when performing scraping tasks.

**1. Legal and Ethical Issues:**

* **Terms of Service (ToS):** Many websites prohibit scraping in their terms of service. Scraping in violation of these terms can lead to legal action or the scraper being blocked from accessing the website.
* **Copyright and Data Ownership:** Data on websites may be protected under copyright law, and scraping this data without permission can lead to intellectual property disputes.
* **Robots.txt and Crawling Guidelines:** Websites often include a robots.txt file to dictate which parts of the site are allowed to be scraped. Ignoring this file can lead to ethical and legal problems.

**2. Dynamic Content and JavaScript Rendering:**

* **JavaScript-Loaded Content:** Many modern websites rely on JavaScript to load content dynamically (such as when data is loaded via AJAX after the initial page load). Traditional scraping methods using libraries like BeautifulSoup and Requests may fail to capture this data.
* **Handling Dynamic Pages:** To overcome this, tools like Selenium or Playwright can be used to simulate a browser environment, but this adds complexity and increases the resource cost of scraping.

**3. Website Structure Changes:**

* **Frequent HTML Structure Changes:** Websites can change their HTML structure at any time, breaking scrapers that rely on specific tags, IDs, or classes. Maintaining scrapers involves continuous monitoring and updating of code to adapt to these changes.
* **Inconsistent HTML Tags:** Websites may use non-standard, inconsistent, or poorly structured HTML, making it harder to accurately extract data using automated methods.

**4. CAPTCHA and Bot Detection:**

* **CAPTCHA Systems:** Many websites employ CAPTCHA mechanisms to differentiate between bots and humans. CAPTCHAs can halt a scraper's operation unless the CAPTCHA is solved, which is challenging for bots.
* **Anti-Scraping Measures:** Websites often have bot detection algorithms that monitor user behavior. If the scraper sends too many requests in a short period or mimics non-human browsing patterns, the IP address may get blocked or throttled.
* **IP Blocking and Rate Limiting:** Some websites block specific IPs that make numerous requests. Bypassing these restrictions may require the use of rotating proxies or VPNs, adding to the cost and complexity.

**5. Data Quality Issues:**

* **Incomplete or Noisy Data:** The extracted data may be incomplete or contain noise (irrelevant information). Scrapers may pick up advertisements, unrelated text, or broken HTML, resulting in the need for further data cleaning and preprocessing.
* **Encoding and Formatting Issues:** Different web pages might use different encodings (e.g., UTF-8, ISO-8859-1), causing errors in the extraction process. Scraped data may need conversion into a common format to be useful for analysis.

**6. Ethical Considerations and Server Overload:**

* **Server Load and Bandwidth Usage:** Sending numerous requests in a short span of time can overwhelm the target website's server, leading to performance degradation or downtime. Ethical scrapers need to introduce delays between requests and throttle the scraping speed to avoid overloading the server.
* **Respect for Website Bandwidth:** Scrapers should take care to minimize bandwidth usage, especially for websites that may have limited resources.

**7. Data Volatility and Updates:**

* **Real-Time Data Changes:** Some websites update their content frequently (e.g., news sites or stock price listings). Scraping such data requires careful scheduling and monitoring to ensure the scraper runs at appropriate intervals to capture the latest information without missing updates.
* **Consistency Over Time:** Due to the dynamic nature of some websites, the scraped data might change frequently, making it challenging to maintain consistency and continuity in the collected data.

**8. Data Privacy and Regulations:**

* **Personal Data Protection:** Many websites contain personal information or sensitive data protected by privacy laws like GDPR (General Data Protection Regulation) and CCPA (California Consumer Privacy Act). Scraping such data without permission can lead to significant legal consequences.
* **Regulatory Compliance:** Scrapers need to ensure that they comply with data protection regulations, which can limit the scope of the data they are legally allowed to scrape.

1. Demonstrate a python web-scraping script using beutifulsoup library for navigating any html f ile of your choice.

### Python Web Scraping Script using BeautifulSoup (10 Marks)

**Introduction:** BeautifulSoup is a Python library that allows easy navigation, searching, and modification of HTML and XML documents. It is particularly useful for web scraping as it helps parse HTML files and extract the desired data. The script below demonstrates the basic use of BeautifulSoup to extract information from an HTML file.

**Sample HTML File:**

For this demonstration, let’s assume we are working with a simple HTML file named example.html that contains information about some books:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Book Store</title>

</head>

<body>

<h1>Available Books</h1>

<div class="book" id="book1">

<h2>Title: Python for Beginners</h2>

<p>Author: John Doe</p>

<p>Price: $30</p>

</div>

<div class="book" id="book2">

<h2>Title: Data Science with Python</h2>

<p>Author: Jane Smith</p>

<p>Price: $45</p>

</div>

<div class="book" id="book3">

<h2>Title: Machine Learning Basics</h2>

<p>Author: Mike Brown</p>

<p>Price: $50</p>

</div>

</body>

</html>

**Python Web Scraping Script:**

This script uses the BeautifulSoup library to navigate and extract data (book titles, authors, and prices) from the HTML file example.html.

# Importing required libraries

from bs4 import BeautifulSoup

# Load the HTML content (In a real-world case, this would be fetched via an HTTP request)

with open('example.html', 'r', encoding='utf-8') as file:

html\_content = file.read()

# Parse the HTML using BeautifulSoup

soup = BeautifulSoup(html\_content, 'html.parser')

# Extract and print the title of the webpage

page\_title = soup.title.text

print("Page Title:", page\_title)

# Find all book divs

books = soup.find\_all('div', class\_='book')

# Iterate over each book and extract information

for book in books:

# Extract and print the book title

title = book.find('h2').text

print("Book Title:", title)

# Extract and print the book author

author = book.find('p').text

print("Author:", author)

# Extract and print the book price

price = book.find\_all('p')[1].text

print("Price:", price)

print('---')

**Explanation of the Script:**

1. **Loading the HTML File:**
   * The HTML content is loaded from a local file example.html. In a real-world scenario, the HTML would typically be fetched from a website using a library like requests, but for simplicity, we use a local file here.
2. **Parsing the HTML:**
   * The BeautifulSoup library is used to parse the HTML content. The parser type html.parser is chosen, but other parsers like lxml or html5lib can also be used.
3. **Extracting the Page Title:**
   * The soup.title.text extracts the title of the web page, which in this case is "Book Store".
4. **Finding Specific Elements:**
   * The script searches for all <div> elements with the class book. Each <div> contains information about one book (title, author, and price).
5. **Navigating and Extracting Data:**
   * For each book, the script extracts the title (<h2> tag), author (first <p> tag), and price (second <p> tag). The .find() method is used for finding the first occurrence of a tag, while .find\_all() is used for multiple occurrences.
6. **Output:**
   * The information is printed for each book, providing details like book title, author, and price.

Page Title: Book Store

Book Title: Title: Python for Beginners

Author: Author: John Doe

Price: Price: $30

---

Book Title: Title: Data Science with Python

Author: Author: Jane Smith

Price: Price: $45

---

Book Title: Title: Machine Learning Basics

Author: Author: Mike Brown

Price: Price: $50

---

1. Illustrate the difference between loc and iloc function in Python program with a code snippet.

In Python, loc and iloc are functions provided by the Pandas library to access data from a DataFrame. They serve similar purposes but differ in the way they select data:

* **loc** is used for label-based indexing. It allows you to select rows and columns by labels (row/column names or indices).
* **iloc** is used for position-based indexing. It selects rows and columns by integer positions (row/column index numbers).

Here’s a detailed illustration with a code snippet:

**Code Snippet Demonstrating loc and iloc:**

**import pandas as pd**

**# Create a sample DataFrame**

**data = {**

**'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],**

**'Age': [25, 30, 35, 40, 45],**

**'Score': [85, 90, 95, 80, 75]**

**}**

**df = pd.DataFrame(data, index=['a', 'b', 'c', 'd', 'e'])**

**# Display the DataFrame**

**print("Original DataFrame:\n", df)**

**# Using loc (label-based indexing)**

**print("\nUsing loc to select row 'b' and columns 'Name' and 'Age':")**

**print(df.loc['b', ['Name', 'Age']])**

**# Using iloc (position-based indexing)**

**print("\nUsing iloc to select the second row (index 1) and first two columns:")**

**print(df.iloc[1, [0, 1]])**

**# Using loc to select multiple rows and columns**

**print("\nUsing loc to select rows 'c' to 'e' and columns 'Name' and 'Score':")**

**print(df.loc['c':'e', ['Name', 'Score']])**

**# Using iloc to select a range of rows and columns by position**

**print("\nUsing iloc to select rows 2 to 4 and columns 0 and 2:")**

**print(df.iloc[2:5, [0, 2]])**

**Explanation:**

1. **DataFrame Creation:**
   * A sample DataFrame is created with rows labeled as a, b, c, d, e and columns Name, Age, and Score.
2. **Using loc:**
   * **Label-based indexing:** We use row and column labels to access data.
   * Example: df.loc['b', ['Name', 'Age']] selects the row labeled 'b' and the columns 'Name' and 'Age'.
3. **Using iloc:**
   * **Position-based indexing:** We use row and column positions (integer index) to access data.
   * Example: df.iloc[1, [0, 1]] selects the second row (index 1) and the first two columns (positions 0 and 1).
4. **Ranges in loc and iloc:**
   * **Label-based range selection:** df.loc['c':'e', ['Name', 'Score']] selects rows 'c' to 'e' and columns 'Name' and 'Score'.
   * **Position-based range selection:** df.iloc[2:5, [0, 2]] selects rows from position 2 to 4 and columns at positions 0 and 2.

**Original DataFrame:**

**Name Age Score**

**a Alice 25 85**

**b Bob 30 90**

**c Charlie 35 95**

**d David 40 80**

**e Eve 45 75**

**Using loc to select row 'b' and columns 'Name' and 'Age':**

**Name Bob**

**Age 30**

**Name: b, dtype: object**

**Using iloc to select the second row (index 1) and first two columns:**

**Name Bob**

**Age 30**

**Name: b, dtype: object**

**Using loc to select rows 'c' to 'e' and columns 'Name' and 'Score':**

**Name Score**

**c Charlie 95**

**d David 80**

**e Eve 75**

**Using iloc to select rows 2 to 4 and columns 0 and 2:**

**Name Score**

**c Charlie 95**

**d David 80**

**e Eve 75**

**Key Differences:**

* **loc:** Uses **labels** (row/column names or indices) to select data.
* **iloc:** Uses **integer positions** (row/column index numbers) to select data.

1. Write a python program to create 3x4 array, delete 3rd column and insert new column in the 3rd column. Print the intermediate results.

**import numpy as np**

**# Step 1: Create a 3x4 array**

**array = np.array([[1, 2, 3, 4],**

**[5, 6, 7, 8],**

**[9, 10, 11, 12]])**

**print("Original Array (3x4):")**

**print(array)**

**# Step 2: Delete the 3rd column (index 2)**

**array\_without\_3rd\_col = np.delete(array, 2, axis=1)**

**print("\nArray after deleting 3rd column:")**

**print(array\_without\_3rd\_col)**

**# Step 3: Create a new column to insert (3 elements)**

**new\_column = np.array([[13], [14], [15]])**

**# Step 4: Insert the new column in the 3rd column position (index 2)**

**array\_with\_new\_col = np.insert(array\_without\_3rd\_col, 2, new\_column, axis=1)**

**print("\nArray after inserting new column at 3rd column position:")**

**print(array\_with\_new\_col)**

**Explanation:**

1. **Creating the Array:**
   * We use np.array() to create a 3x4 array with specific values.
2. **Deleting the 3rd Column:**
   * The np.delete() function is used to remove the column at index 2 (3rd column), and axis=1 specifies that we're working with columns.
   * The result is stored in array\_without\_3rd\_col.
3. **Creating a New Column:**
   * A new column with 3 elements is created as a 3x1 array using np.array().
4. **Inserting the New Column:**
   * We use np.insert() to insert the new column at index 2 (3rd column position). Again, axis=1 is used to specify column insertion.

**Original Array (3x4):**

**[[ 1 2 3 4]**

**[ 5 6 7 8]**

**[ 9 10 11 12]]**

**Array after deleting 3rd column:**

**[[ 1 2 4]**

**[ 5 6 8]**

**[ 9 10 12]]**

**Array after inserting new column at 3rd column position:**

**[[ 1 2 13 4]**

**[ 5 6 14 8]**

**[ 9 10 15 12]]**

**Key Functions:**

* **np.array()**: Creates a NumPy array.
* **np.delete()**: Removes a specified column or row from an array.
* **np.insert()**: Inserts a new column or row at a specific position in the array.