1. Write Python code for extracting Title and href attributes from a HTML document?

```
import requests
from bs4 import BeautifulSoup
r=requests.get(“https.//www.google.com)
print(r.text)
soup=BeautifulSoup(r.text,”html5lib”)
s=soup.title.string
if s:
  print(“title:”,s)
else:
 print(“not found”)
href=soup.findall(‘a’,href=True)
if href:
  print(“hrefs:”,href)
else:
  print(“no anchor tags”)
```

2. Write a Python function that uses CSS selectors to extract all links from a given webpage.

```
import requests
from bs4 import BeautifulSoup
def extract_links(url):
  r=response.get(url)
  soup=BeautifulSoup(r.text,”html5lib”)
  link=soup.select(‘a[href]’)
  return link
url=”https.//www.google.com
link=extract_links(url)
print(“Extracted links:”,link)
```

3. Describe the process of making an HTTP GET request in Python using the requests library and print the response status code and its content.

   - To make an HTTP GET request, we need use the requests library. If we don’t have the requests library than we need to install it by using pip which is a python package manager.
     Syntax: pip install requests
   - After installing the requests library, we make an HTTP GET request by using a function called requests.get()
   - After making a GET request ,we can get the status code and content in the form of .content and .status_code.

Code for the above process:

```
import requests
r=requests.get("https.www.google.com)
print(r.content)
print(r.status_code)
```

4. How would you handle session cookies while scraping a website that requires login?
   - When scraping a website that requires login, handling session cookies is essential to maintain an authenticated session. You can do this in Python using the requests.Session object, which allows you to persist cookies across multiple requests.
   - **Create a Session Object**: Using `requests.Session()`, you can maintain session cookies across multiple requests.
   - **Send Login Request**: Send a POST request to the login URL with the required credentials. This will store the session cookie in the `Session` object.
   - **Scrape Authenticated Pages**: Use the same session object to make additional requests to other pages that require authentication. The session will automatically handle the cookies set during login.
   - **Handle CSRF Tokens** *(Optional)*: Some websites require a CSRF (Cross-Site Request Forgery) token along with credentials. Often, you'll need to scrape the token from the login page before making the login request.

5. How does Scrapy handle requests and responses?

   Spider :generates requests and sets callback functions for handling responses.
   Scheduler :queues requests for crawling and prioritizes them.
   Downloader :fetches the requested pages, respecting site policies.
   Response :is processed in the spider's callback function for data extraction.
   Pipelines :process extracted data, preparing it for storage or further use.
   Middleware :layers enable custom handling of requests and responses.

6. What is Scrapy, and how does it differ from other web scraping libraries like BeautifulSoup?
   - **Scrapy** is an open-source and versatile web scraping and web crawling framework specifically designed for extracting data from websites. It enables users to efficiently scrape large amounts of data from websites, automatically handling multiple requests and responses, downloading pages, and managing data extraction. Scrapy is particularly suitable for more complex scraping tasks and is widely used in production environments due to its high speed and scalability.

     Scrapy and **BeautifulSoup** are both popular tools for web scraping, but they serve different purposes and have distinct strengths.

Primary Use Case:

- **Scrapy** is a full-fledged scraping framework, ideal for large-scale scraping tasks and recursive crawls. It's designed to handle complex data extraction and web crawling, making it a strong choice for production-level scraping.
- **BeautifulSoup**, on the other hand, is primarily an HTML parser used for smaller, simpler scraping projects. It's well-suited for parsing static HTML content where only a few specific elements need to be extracted.

Concurrency:

- **Scrapy** has built-in asynchronous processing (using Twisted), which allows it to handle multiple requests simultaneously. This makes it extremely fast and efficient, especially when dealing with large sites.
- **BeautifulSoup** operates synchronously and handles one request at a time. It doesn't have built-in support for concurrent requests, making it slower for high-volume scraping tasks.

Crawling Capabilities:

- **Scrapy** has native support for crawling, with features that allow it to follow links, handle pagination, and automatically download pages. It's highly efficient for deep web scraping where multiple pages need to be traversed.
- **BeautifulSoup** lacks these crawling capabilities. It only parses HTML from provided content, meaning that to achieve similar crawling functionality, users need to pair it with other libraries like `requests` for handling HTTP requests and following links.

Data Export Options:

- **Scrapy** has built-in support for exporting data to formats like JSON, CSV, or even databases. It's equipped with pipelines that make it easy to process, clean, and store scraped data.
- **BeautifulSoup** doesn't have any native export options. Users have to manually handle data export, typically by writing data to files or databases themselves.

Request Management:

- **Scrapy** directly manages requests, allowing users to customize headers, handle cookies, use proxies, and implement retry logic. This makes it highly adaptable to various scraping requirements, including those that need session management.
- **BeautifulSoup** does not handle requests directly. It relies on external libraries, like `requests`, to retrieve HTML, after which it parses the content. This adds an extra layer of complexity for handling headers, cookies, or retries.

➢ **Performance**:

   - **Scrapy** is highly efficient and performs well for scraping large or dynamic sites, thanks to its asynchronous request processing.
   - **BeautifulSoup** is generally slower for high-volume scraping, as it handles each request sequentially. It's better suited for smaller, static scraping tasks.

7. How would you handle complex scraping projects with multiple spiders and data pipelines in Scrapy?
   - ➢ In Scrapy, handle complex scraping projects by creating **multiple spiders** to target different sections or patterns of the website, each tailored to specific data extraction needs. Use **item pipelines** to process, clean, and store the scraped data, applying separate stages for validation, transformation, and export to databases or files. For efficient management, organize spiders and pipelines in a modular structure, and configure settings for each spider, like concurrency limits and delays, to optimize performance and avoid getting blocked.
8. Write a Python script that scrapes the title, headings (h1, h2, etc.), and paragraphs (p) from a given webpage URL and outputs the following:

   The title of the page

   All the headings in the page (h1, h2, h3, etc.)

   All the paragraphs

   Example:

   - URL: https://www.example.com
   - Output:
     - Title: "Example Domain"
     - Headings: ['Heading 1', 'Heading 2']
     - Paragraphs: ['This is a paragraph.']

```
import requests
from bs4 import BeautifulSoup

def scrape_webpage(url):
  response = requests.get(url)
  if response.status_code != 200:
    print(f"Failed to retrieve page. Status code: {response.status_code}")
    return

  soup = BeautifulSoup(response.text, 'html.parser')

  title = soup.title.string if soup.title else "No title found"

  headings = []
```

```
    for level in range(1, 7):
        for tag in soup.find_all(f'h{level}'):
            headings.append(tag.get_text(strip=True))

    paragraphs = [p.get_text(strip=True) for p in soup.find_all('p')]



    print("URL:", url)
    print("Title:", title)
    print("Headings:", headings)
    print("Paragraphs:", paragraphs)

url = 'https://www.example.com'
scrape_webpage(url)
```

9. Write a Python script to scrape a website with multiple pages (e.g., an e-commerce site or a blog with paginated articles). Your task is to:

   Extract the titles of articles/products.

   Iterate through multiple pages to get all the data. (Assume the pagination is implemented using "Next" buttons or page numbers in the URL).

   Example:

- URL: https://example.com/products?page=1
- Expected Output:
  - Titles of products from pages 1, 2, and 3.

```
import requests
from bs4 import BeautifulSoup

def scrape_titles(base_url, max_pages=3):
  page = 1
   titles = []

   while page <= max_pages:
       url = f"{base_url}?page={page}"
      response = requests.get(url)

       if response.status_code != 200:
         print(f"Failed to retrieve page {page}. Status code:
{response.status_code}")
        break
  soup = BeautifulSoup(response.text, 'html.parser')
        page_titles = [title.get_text(strip=True) for title in soup.select('h2.title')]
     if not page_titles:  # Stop if no titles are found, indicating there are no more
pages
         break

    titles.extend(page_titles)
        print(f"Scraped {len(page_titles)} titles from page {page}.")
       page += 1

  return titles

base_url = 'https://example.com/products'
titles = scrape_titles(base_url, max_pages=3)
print("Extracted Titles:", titles)
```

10. Write a Python script that scrapes product prices for the same product from multiple e-commerce websites (e.g., Amazon, Flipkart, TataCliQ).

    Your task is to:

    Search for the same product across multiple websites (e.g., "laptop").

    Extract product name, price, and rating (if available).

    Output the results in a structured format like a CSV or JSON.

    Example:

- Search term: "Laptop"
- Expected Output:

```
[
  {
    "product_name": "Laptop 1",
    "price": "$500",
    "rating": "4.5/5",
    "website": "amazon.com"
  },
  {
    "product_name": "Laptop 2",
    "price": "$450",
    "rating": "4.0/5",
    "website": "tatacliq.com"
  }
]
```

```
import requests
from bs4 import BeautifulSoup
import json
```

```
def scrape_site(search_term, base_url, query_param, selectors, website):
    url = f"{base_url}{query_param}{search_term.replace(' ', '+')}"
    headers = {'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/90.0.4430.85
Safari/537.36'}
    response = requests.get(url, headers=headers)

    if response.status_code != 200:
        print(f"Failed to retrieve results from {website}")
        return []

    soup = BeautifulSoup(response.text, 'html.parser')
    products = []

    for item in soup.select(selectors['product_container']):
        title = item.select_one(selectors['title']).get_text(strip=True) if
item.select_one(selectors['title']) else "N/A"
        price = item.select_one(selectors['price']).get_text(strip=True) if
item.select_one(selectors['price']) else "N/A"
        rating = item.select_one(selectors['rating']).get_text(strip=True) if
item.select_one(selectors['rating']) else "No rating"

        products.append({
            "product_name": title,
            "price": price,
            "rating": rating,
            "website": website
        })

    return products

def scrape_all_sites(search_term):
    ecomm_sites = [
        {
            "base_url": "https://www.amazon.com/s?k=",
            "query_param": "",
            "selectors": {
                "product_container": ".s-result-item",
                "title": "h2 a span",
                "price": ".a-price-whole",
                "rating": ".a-icon-alt"
            },
            "website": "amazon.com"
        },
        {
            "base_url": "https://www.flipkart.com/search?q=",
            "query_param": "",
            "selectors": {
                "product_container": "._1AtVbE",
                "title": "._4rR01T",
                "price": "._30jeq3",
                "rating": "._3LWZlK"
            },
            "website": "flipkart.com"
        },
        {
            "base_url": "https://www.tatacliq.com/search/?text=",
            "query_param": "",
            "selectors": {
                "product_container": ".ProductModule__productName",
                "title": ".ProductModule__productName",
                "price": ".Price__basePrice",
                "rating": ".ProductRating__rating"
            },
            "website": "tatacliq.com"
        }
    ]

    all_products = []
    for site in ecomm_sites:
        all_products.extend(scrape_site(search_term,                site["base_url"],
site["query_param"], site["selectors"], site["website"]))


    with open(f'{search_term}_products.json', 'w', encoding='utf-8') as file:
        json.dump(all_products, file, indent=4, ensure_ascii=False)
    print(f"Data saved to {search_term}_products.json")

search_term = "Laptop"
scrape_all_sites(search_term)
```

11. Write a Python script that scrapes weather data for a given location from a weather website (e.g., Weather.com). You need to extract:
    - Current temperature.
    - Weather conditions (sunny, rainy, etc.).
    - 7-day forecast (if available).

      Example Url: https://weather.com/weather/today/l/<Country_name>

```
import requests
from bs4 import BeautifulSoup

def scrape_weather(location_url):
    response = requests.get(location_url)
    if response.status_code != 200:
        print(f"Failed to retrieve weather data. Status code: {response.status_code}")
        return

    soup = BeautifulSoup(response.text, 'html.parser')

    temperature = soup.find('span', class_='CurrentConditions--tempValue--
3KcTQ').get_text() if soup.find('span', class_='CurrentConditions--tempValue--
3KcTQ') else "N/A"

    condition = soup.find('div', class_='CurrentConditions--phraseValue--
2xXSr').get_text() if soup.find('div', class_='CurrentConditions--phraseValue--
2xXSr') else "N/A"

    forecast = []
    daily_forecast = soup.find_all('details', class_='DaypartDetails--DayPartDetail--
3yhtR')
    for day in daily_forecast[:7]:  # Limit to 7 days if there are more entries
        day_name = day.find('h3').get_text(strip=True) if day.find('h3') else "N/A"
        day_temp = day.find('span', class_='DetailsSummary--highTempValue--
3x6cL').get_text(strip=True) if day.find('span', class_='DetailsSummary--
highTempValue--3x6cL') else "N/A"
        night_temp = day.find('span', class_='DetailsSummary--lowTempValue--
1DlJK').get_text(strip=True) if day.find('span', class_='DetailsSummary--
lowTempValue--1DlJK') else "N/A"
        forecast.append({
            'day': day_name,
            'day_temp': day_temp,
            'night_temp': night_temp
        })

    print(f"Location URL: {location_url}")
    print(f"Current Temperature: {temperature}")
    print(f"Current Condition: {condition}")
    print("7-Day Forecast:")
    for day in forecast:
        print(f"  {day['day']}: Day Temp: {day['day_temp']}, Night Temp:
{day['night_temp']}")

location_url = 'https://weather.com/weather/today/l/USCA0286:1:US'  # Replace
with the actual location URL
scrape_weather(location_url)
```

12. Write a Python script to scrape job listings from a job board (e.g., Indeed, LinkedIn). You need to:

- Extract job titles, company names, locations, and the job descriptions.
- Filter the results by a specific keyword (e.g., “data scientist”).
- Store the results in a CSV or JSON file.

Example:

- Input keyword: “data scientist”
- Output:

job_title, company, location, job_description

"Data Scientist", "ABC Corp", "Hyderabad, India", "Looking for a data scientist with experience in machine learning..."

```
import requests

from bs4 import BeautifulSoup

import csv

import json

def scrape_jobs(base_url, keyword, max_pages=1, output_file='jobs.csv'):

   jobs = []

   for page in range(1, max_pages + 1):

      url = f"{base_url}?q={keyword}&start={page * 10}"

      response = requests.get(url)

      if response.status_code != 200:

         print(f"Failed to retrieve page {page}. Status code: {response.status_code}")

         break

        soup = BeautifulSoup(response.text, 'html.parser'

      job_listings = soup.find_all('div', class_='jobsearch-SerpJobCard')  # Example for Indeed

      for job in job_listings:
         job_title = job.find('h2', class_='title').get_text(strip=True) if job.find('h2', class_='title') else "N/A"

         company = job.find('span', class_='company').get_text(strip=True) if job.find('span', class_='company') else "N/A"

         location = job.find('div', class_='location').get_text(strip=True) if job.find('div', class_='location') else "N/A"

         job_description = job.find('div', class_='summary').get_text(strip=True) if job.find('div', class_='summary') else "N/A"



         jobs.append({

            'job_title': job_title,

            'company': company,

            'location': location,

            'job_description': job_description

         })

      print(f"Scraped {len(job_listings)} jobs from page {page}.")

   with open(output_file, 'w', newline='', encoding='utf-8') as file:

      writer = csv.DictWriter(file, fieldnames=['job_title', 'company', 'location', 'job_description'])

      writer.writeheader()

      writer.writerows(jobs)

  json_output_file = output_file.replace('.csv', '.json')

   with open(json_output_file, 'w', encoding='utf-8') as json_file:

      json.dump(jobs, json_file, indent=4, ensure_ascii=False)

   print(f"Data saved to {output_file} and {json_output_file}")



keyword = "data scientist"
```

```
base_url = 'https://www.indeed.com/jobs'

scrape_jobs(base_url, keyword, max_pages=2, output_file='data_scientist_jobs.csv')
```