**Web Scraping**

A web scraper is a program that can extract data from websites and save it in a structured format, such as a CSV or JSON file. Web scraping can be useful for a variety of tasks, such as data mining, data analysis, and price comparison.

**Introduction to Web Scaping**

Web scraping is a technique to fetch data from websites. While surfing on the web, many websites don’t allow the user to save data for personal use. One way is to manually copy-paste the data, which both tedious and time-consuming. Web Scraping is the automation of the data extraction process from websites. This event is done with the help of web scraping software known as web scrapers. They automatically load and extract data from the websites based on user requirements. These can be custom built to work for one site or can be configured to work with any website.

**Uses of Web Scraping:** Web scraping finds many uses both at a professional and personal level. Having different needs at different levels, some popular uses of web scraping are.

* **Brand Monitoring and Competition Analysis:** Web Scraping is used to get customer feedback regarding a particular service or product to understand how a customer feels regarding that particular thing. It is also used to extract competitor data in a structural, usable format.
* **Machine Learning:** Machine Learning is a process of Artificial Intelligence in which the machine is allowed to learn and improve with its experience rather than being explicitly programmed. For that, a large amount of data is required from millions of sites which is extracted through web scraping software.
* **Financial Data Analysis:** Web Scraping is used to keep a record of the stock market in a usable format and hence employ the same for insights.
* **Social Media Analysis:** It is used to extract data from social media sites to gauge customer trends, and how they react to the campaign.
* **SEO monitoring:** Search Engine Optimization is the optimization of the visibility and ranking of a website among different search engines like Google, Yahoo, Bing, etc. Web scraping is used to understand how the ranking of the content over time.

And there are so many other reasons to use Web Scrapping.

**Techniques of Web Scraping:** There are two ways of extracting data from websites, the Manual extraction technique, and the automated extraction technique.

**Manual Extraction Techniques:** Manually copy-pasting the site content comes under this technique. Though tedious, time taking and repetitive it is an effective way to scrap data from the sites having good anti-scraping measures like bot detection.

**Automated Extraction Techniques:** Web scraping software is used to automatically extract data from sites based on user requirement.

* **HTML Parsing:** Parsing means to make something understandable to be analysing it part by part. To wit, it means to convert the information in one form to another form that is easy to that is easier to work on with. HTML parsing means taking in the code and extracting relevant information from it based on the user requirement. Mainly executed using JavaScript, the target as the name suggests are HTML pages.
* **DOM Parsing:** The Document Object Model is the official recommendation of the World Wide Web Consortium. It defines an interface that enables a user to modify and update the style, structure, and content of the XML document.
* **Web Scraping Software:** Nowadays, many web scraping tools are available or are custom build on users need to extract required desiring information from millions of websites.

Tool for Web Scraping: Web Scraping tools are specifically developed for extracting data from the internet. Also, known as web harvesting tools or data extraction tools, they are useful for anyone trying to collect specific data from websites as they provide the user with structured data extracting data from several websites. Some of the most popular Web Scraping tools are:

* Import.io
* Webhose.io
* Dexi.io
* Scrapinghub
* Parsehub

**Legalization of Web Scraping:** The legalization of web scraping is a sensitive topic, depending on how it is used it can either be a boon or a bane. On one hand, web scraping with good bot enables search engines to index web content, price comparison services to save customer money and value. But web scraping can be re-targeted to meet more malicious and abusive ends. Web scraping can be aligned with other forms of malicious automation, named “bad bots”, which enable other harmful activities like denial-of-service attacks, competitive data mining, account hijacking, data theft etc.

Legality of Web Scraping is a grey area that tends to develop as time goes on. Although the web scrapers technically increase the speed up data surfing, loading, copying, and pasting web scraping is also the key culprit behind the increase’s cases of copyright violation, violated terms of use and other activities that are highly disruptive to a company’s business.

**Challenges to Web Scraping:** Besides the challenge of the legality of web scraping, there are also other problems that pose a challenge to web scraping.

* **Data Warehousing:** Data extraction at a scale will generate a large amount of information to be stored. If the data warehousing infrastructure is not properly built then the searching, storing and exporting of this data will become a cumbersome task. Hence, for large-scale data extraction, there needs to be a perfect data warehousing system without any flaws and faults.
* **Website Structure Changes:** Every website periodically updates its user interface to improve its attractiveness and experience. This requires various structural changes too. Since the web scrapers are set up according to the code elements of the website at that time, they require changes too. So, they require changes weekly too to target the correct website for data scraping as incomplete information regarding the website structure will lead to improper scraping of data.
* **Anti-Scraping Technologies:** Some websites use anti-scraping technologies that thwart away any scraping attempt. They apply a dynamic coding algorithm to prevent any bot intervention and use the IP blocking mechanism. It requires a lot of time and money to work around such anti-scraping technologies.
* **Quality of Data Extracted:** Records that do not meet the quality of information required will affect the overall integrity of the data. Making sure that the Data Scraped meets the quality guidelines is a difficult task as it needs to be done in real-time.

**Future of Data Scraping:** As there are some challenges and opportunities for data scraping, it can be fairly deemed that the unintended data-scraping practitioners are prone to create a moral hazard where they target the companies and retrieve their data. However, since we are on the verge of data transformation, data-scraping in combination with big data can provide the company’s market intelligence and help them identify critical trends and patterns and identify the best opportunities and solutions. Hence, it won’t be wrong to say that Data scraping can be upgraded to the better soon.

**What is Web Scraping and How to Use It?**

Suppose you want some information from a website? Let’s say a paragraph on Donald Trump! What do you do? Well, you can copy and paste the information from Wikipedia to your own file. But what if you want to get large amounts of information from a website as quickly as possible? Such as large amounts of data from a website to train a [Machine Learning](https://www.geeksforgeeks.org/machine-learning/) algorithm? In such a situation, copying and pasting will not work! And that’s when you’ll need to use **Web Scraping**.



Unlike the long and mind-numbing process of manually getting data, Web scraping uses intelligence automation methods to get thousands or even millions of data sets in a smaller amount of time. So, let’s understand what Web scraping is in detail and how to use it to obtain data from other websites.

**What is Web Scraping?**

Web scraping is an automatic method to obtain large amounts of data from websites. Most of this data is unstructured data in an HTML format which is then converted into structured data in a spreadsheet or a database so that it can be used in various applications. There are many ways to perform web scraping to obtain data from websites. These include using online services, particular API’s or even creating your code for web scraping from scratch. Many large websites, like Google, Twitter, Facebook, Stack Overflow, etc. have API’s that allow you to access their data in a structured format. This is the best option, but there are other sites that don’t allow users to access large amounts of data in a structured form or they are simply not that technologically advanced. In that situation, it’s best to use Web Scraping to scrape the website for data.

Web scraping requires two parts, namely the **crawler** and the **scraper**. The crawler is an artificial intelligence algorithm that browses the web to search for the data required by following the links across the internet. The scraper, on the other hand, is a specific tool created to extract data from the website. The design of the scraper can vary greatly according to the complexity and scope of the project so that it can quickly and accurately extract the data.

**How Web Scrapers Work?**

Web Scrapers can extract all the data on sites or the specific data that a user wants. Ideally, it’s best if you specify the data, you want so that the web scraper only extracts that data quickly. For example, you might want to scrape an Amazon page for the types of juicers available, but you might only want the data about the models of different juicers and not the customer reviews.

So, when a web scraper needs to scrape a site, first the URLs are provided. Then it loads all the HTML code for those sites and a more advanced scraper might even extract all the CSS and JavaScript elements as well. Then the scraper obtains the required data from this HTML code and outputs this data in the format specified by the user. Mostly, this is in the form of an Excel spreadsheet or a CSV file, but the data can also be saved in other formats, such as a JSON file.

**Different Types of Web Scrapers**

Web Scrapers can be divided based on many different criteria, including Self-built or Pre-built Web Scrapers, Browser extension or Software Web Scrapers, and Cloud or Local Web Scrapers.

You can have **Self-built Web Scrapers** but that requires advanced knowledge of programming. And if you want more features in your Web Scrapper, then you need even more knowledge. On the other hand, pre-built**Web Scrapers** are previously created scrapers that you can download and run easily. These also have more advanced options that you can customize.

**Browser extensions Web Scrapers** are extensions that can be added to your browser. These are easy to run as they are integrated with your browser, but at the same time, they are also limited because of this. Any advanced features that are outside the scope of your browser are impossible to run on Browser extension Web Scrapers. But **Software Web Scrapers** don’t have these limitations as they can be downloaded and installed on your computer. These are more complex than Browser web scrapers, but they also have advanced features that are not limited by the scope of your browser.

**Cloud Web Scrapers** run on the cloud, which is an off-site server mostly provided by the company that you buy the scraper from. These allow your computer to focus on other tasks as the computer resources are not required to scrape data from websites. **Local Web Scrapers**, on the other hand, run on your computer using local resources. So, if the Web scrapers require more CPU or RAM, then your computer will become slow and not be able to perform other tasks.

**Why is Python a popular programming language for Web Scraping?**

[Python](https://www.geeksforgeeks.org/python-programming-language/) seems to be in fashion these days! It is the most popular language for web scraping as it can handle most of the processes easily. It also has a variety of libraries that were created specifically for Web Scraping. [**Scrapy**](https://scrapy.org/) is a very popular open-source web crawling framework that is written in Python. It is ideal for web scraping as well as extracting data using APIs. [**Beautiful soup**](https://pypi.org/project/beautifulsoup4/) is another Python library that is highly suitable for Web Scraping. It creates a parse tree that can be used to extract data from HTML on a website. Beautiful soup also has multiple features for navigation, searching, and modifying these parse trees.

**What is Web Scraping used for?**

Web Scraping has multiple applications across various industries. Let’s check out some of these now!

1. **Price Monitoring**

Web Scraping can be used by companies to scrap the product data for their products and competing products as well to see how it impacts their pricing strategies. Companies can use this data to fix the optimal pricing for their products so that they can obtain maximum revenue.

1. **Market Research**

Web scraping can be used for market research by companies. High-quality web scraped data obtained in large volumes can be very helpful for companies in analysing consumer trends and understanding which direction the company should move in the future.

1. **News Monitoring**

Web scraping news sites can provide detailed reports on the current news to a company. This is even more essential for companies that are frequently in the news or that depend on daily news for their day-to-day functioning. After all, news reports can make or break a company in a single day!

1. **Sentiment Analysis**

If companies want to understand the general sentiment for their products among their consumers, then Sentiment Analysis is a must. Companies can use web scraping to collect data from social media websites such as Facebook and Twitter as to what the general sentiment about their products is. This will help them in creating products that people desire and moving ahead of their competition.

1. **Email Marketing**

Companies can also use Web scraping for email marketing. They can collect Email ID’s from various sites using web scraping and then send bulk promotional and marketing Emails to all the people owning these Email ID’s.

**Implementing Web Scraping in Python With Beautiful Soup**

There are mainly two ways to extract data from a website:

* Use the API of the website (if it exists). For example, Facebook has the Facebook Graph API which allows retrieval of data posted on Facebook.
* Access the HTML of the webpage and extract useful information/data from it. This technique is called web scraping or web harvesting or web data extraction.

This article discusses the steps involved in web scraping using the implementation of a Web Scraping framework of Python called Beautiful Soup. Steps involved in web scraping:

* Send an HTTP request to the URL of the webpage you want to access. The server responds to the request by returning the HTML content of the webpage. For this task, we will use a third-party HTTP library for python-requests.
* Once we have accessed the HTML content, we are left with the task of parsing the data. Since most of the HTML data is nested, we cannot extract data simply through string processing. One needs a parser which can create a nested/tree structure of the HTML data. There are many HTML parser libraries available but the most advanced one is html5lib.
* Now, all we need to do is navigating and searching the parse tree that we created, i.e. tree traversal. For this task, we will be using another third-party python library, Beautiful Soup. It is a Python library for pulling data out of HTML and XML files.

**Step 1: Installing the required third-party libraries**

* Easiest way to install external libraries in python is to use pip. pip is a package management system used to install and manage software packages written in Python. All you need to do is:

pip install requests

pip install html5lib

pip install bs4

* Another way is to download them manually from these links:
* requests
* html5lib
* beautifulsoup4

**Step 2: Accessing the HTML content from webpage**

**import** requests

URL **=** "<https://www.geeksforgeeks.org/data-structures/>"

r **=** requests.get(URL)

print(r.content)

Let us try to understand this piece of code.

* First of all import the requests library.
* Then, specify the URL of the webpage you want to scrape.
* Send a HTTP request to the specified URL and save the response from server in a response object called r.
* Now, as print r.content to get the **raw HTML content** of the webpage. It is of ‘string’ type.

Note: Sometimes you may get error “Not accepted” so try adding a browser user agent like below. Find your user agent based on device and browser from here <https://deviceatlas.com/blog/list-of-user-agent-strings>

headers **=** {'User-Agent': "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/42.0.2311.135 Safari/537.36 Edge/12.246"}

# Here the user agent is for Edge browser on windows 10. You can find your browser user agent from the above given link.

r **=** requests.get(url**=**URL, headers**=**headers)

print(r.content)

**Step 3: Parsing the HTML content**

#This will not run on online IDE

**import** requests

**from** bs4 **import** BeautifulSoup

URL **=** "<http://www.values.com/inspirational-quotes>"

r **=** requests.get(URL)

soup **=** BeautifulSoup(r.content, 'html5lib') # If this line causes an error, run 'pip install html5lib' or install html5lib

print(soup.prettify())

A really nice thing about the BeautifulSoup library is that it is built on the top of the HTML parsing libraries like html5lib, lxml, html.parser, etc. So BeautifulSoup object and specify the parser library can be created at the same time. In the example above,

soup = BeautifulSoup(r.content, 'html5lib')

We create a BeautifulSoup object by passing two arguments:

* **r.content:** It is the raw HTML content.
* **html5lib:** Specifying the HTML parser we want to use.

Now soup.prettify() is printed, it gives the visual representation of the parse tree created from the raw HTML content.

**Step 4: Searching and navigating through the parse tree**

Now, we would like to extract some useful data from the HTML content. The soup object contains all the data in the nested structure which could be programmatically extracted. In our example, we are scraping a webpage consisting of some quotes. So, we would like to create a program to save those quotes (and all relevant information about them).

#Python program to scrape website

#and save quotes from website

**import** requests

**from** bs4 **import** BeautifulSoup

**import** csv

URL **=** "<http://www.values.com/inspirational-quotes>"

r **=** requests.get(URL)

soup **=** BeautifulSoup(r.content, 'html5lib')

quotes**=**[]  # a list to store quotes

table **=** soup.find('div', attrs **=** {'id':'all\_quotes'})

**for** row **in** table.findAll('div',

                         attrs **=** {'class':'col-6 col-lg-3 text-center margin-30px-bottom sm-margin-30px-top'}):

    quote **=** {}

    quote['theme'] **=** row.h5.text

    quote['url'] **=** row.a['href']

    quote['img'] **=** row.img['src']

    quote['lines'] **=** row.img['alt'].split(" #")[0]

    quote['author'] **=** row.img['alt'].split(" #")[1]

    quotes.append(quote)

filename **=** 'inspirational\_quotes.csv'

with open(filename, 'w', newline**=**'') as f:

    w **=** csv.DictWriter(f,['theme','url','img','lines','author'])

    w.writeheader()

**for** quote **in** quotes:

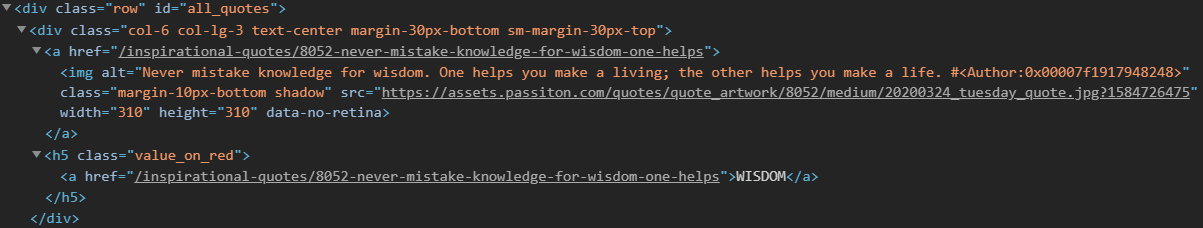
        w.writerow(quote)

Before moving on, we recommend you to go through the HTML content of the webpage which we printed using soup.prettify() method and try to find a pattern or a way to navigate to the quotes.

* It is noticed that all the quotes are inside a div container whose id is ‘all\_quotes’. So, we find that div element (termed as table in above code) using find() method :

table = soup.find('div', attrs = {'id':'all\_quotes'})

* The first argument is the HTML tag you want to search, and second argument is a dictionary type element to specify the additional attributes associated with that tag. find() method returns the first matching element. You can try to print table.prettify() to get a sense of what this piece of code does.
* Now, in the table element, one can notice that each quote is inside a div container whose class is quote. So, we iterate through each div container whose class is quote. Here, we use findAll() method which is similar to find method in terms of arguments but it returns a list of all matching elements. Each quote is now iterated using a variable called row. Here is one sample row HTML content for better understanding:



Now consider this piece of code:

for row in table.find\_all\_next('div', attrs = {'class': 'col-6 col-lg-3 text-center margin-30px-bottom sm-margin-30px-top'}):

quote = {}

quote['theme'] = row.h5.text

quote['url'] = row.a['href']

quote['img'] = row.img['src']

quote['lines'] = row.img['alt'].split(" #")[0]

quote['author'] = row.img['alt'].split(" #")[1]

quotes.append(quote)

* We create a dictionary to save all information about a quote. The nested structure can be accessed using dot notation. To access the text inside an HTML element, we use **.text :**

quote['theme'] = row.h5.text

* We can add, remove, modify and access a tag’s attributes. This is done by treating the tag as a dictionary:

quote['url'] = row.a['href']

* Lastly, all the quotes are appended to the list called **quotes.**
* Finally, we would like to save all our data in some CSV file.

filename = 'inspirational\_quotes.csv'

with open(filename, 'w', newline='') as f:

w = csv.DictWriter(f,['theme','url','img','lines','author'])

w.writeheader()

for quote in quotes:

w.writerow(quote)

* Here we create a CSV file called inspirational\_quotes.csv and save all the quotes in it for any further use.

So, this was a simple example of how to create a web scraper in Python.  From here, you can try to scrap any other website of your choice.

**Web Scraping Tutorial**

Let’s suppose you want to get some information from a website? Let’s say an article from the geeksforgeeks website or some news article, what will you do? The first thing that may come in your mind is to copy and paste the information into your local media. But what if you want a large amount of data on a daily basis and as quickly as possible. In such situations, copy and paste will not work and that’s where you’ll need web scraping.

In this article, we will discuss how to perform web scraping using the requests library and beautifulsoup library in Python.

**Requests Module**

Requests library is used for making HTTP requests to a specific URL and returns the response. Python requests provide inbuilt functionalities for managing both the request and response.

**Installation**

Requests installation depends on the type of operating system, the basic command anywhere would be to open a command terminal and run,

pip install requests

**Making a Request**

Python requests module has several built-in methods to make HTTP requests to specified URI using GET, POST, PUT, PATCH, or HEAD requests. A HTTP request is meant to either retrieve data from a specified URI or to push data to a server. It works as a request-response protocol between a client and a server. Here we will be using the GET request.

GET method is used to retrieve information from the given server using a given URI. The GET method sends the encoded user information appended to the page request.

**Example: Python requests making GET request**

**import** requests

# Making a GET request

r **=** requests.get('<https://www.geeksforgeeks.org/python-programming-language/>')

# check status code for response received

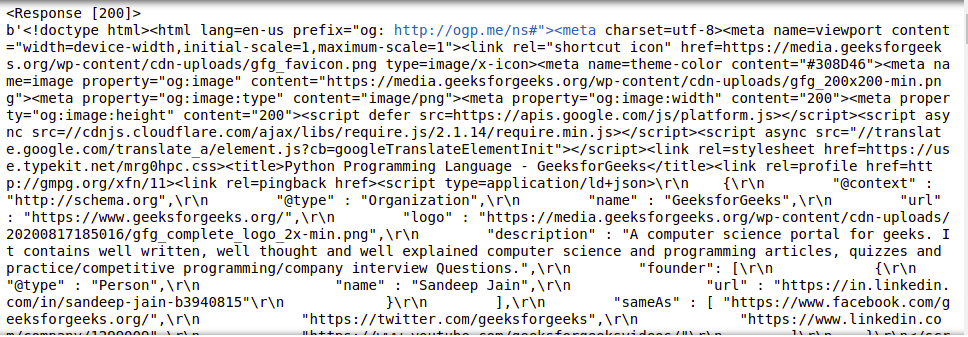
# success code - 200

print(r)

# print content of request

**print**(r.content)

**Output:**



**Response object**

When one makes a request to a URI, it returns a response. This Response object in terms of python is returned by requests.method(), method being – get, post, put, etc. Response is a powerful object with lots of functions and attributes that assist in normalizing data or creating ideal portions of code. For example, response.status\_code returns the status code from the headers itself, and one can check if the request was processed successfully or not.

Response objects can be used to imply lots of features, methods, and functionalities.

**Example: Python requests Response Object**

**import** requests

# Making a GET request

r **=** requests.get('<https://www.geeksforgeeks.org/python-programming-language/>')

# print request object

print(r.url)

# print status code

**print**(r.status\_code)

**Output:**

https://www.geeksforgeeks.org/python-programming-language/

200

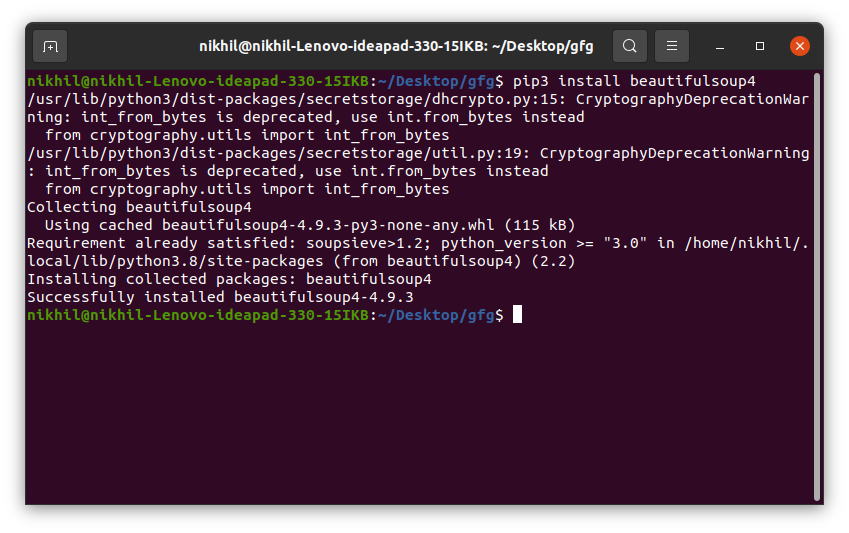
**BeautifulSoup Library**

BeautifulSoup is used extract information from the HTML and XML files. It provides a parse tree and the functions to navigate, search or modify this parse tree.

**Installation**

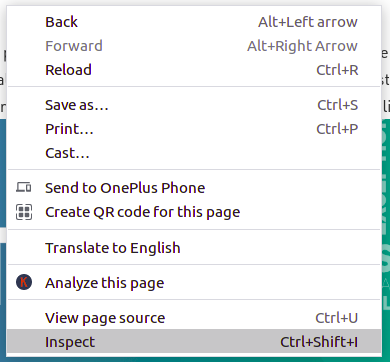
To install Beautifulsoup on Windows, Linux, or any operating system, one would need pip package. To check how to install pip on your operating system, check out – PIP Installation – Windows || Linux. Now run the below command in the terminal.

pip install beautifulsoup4

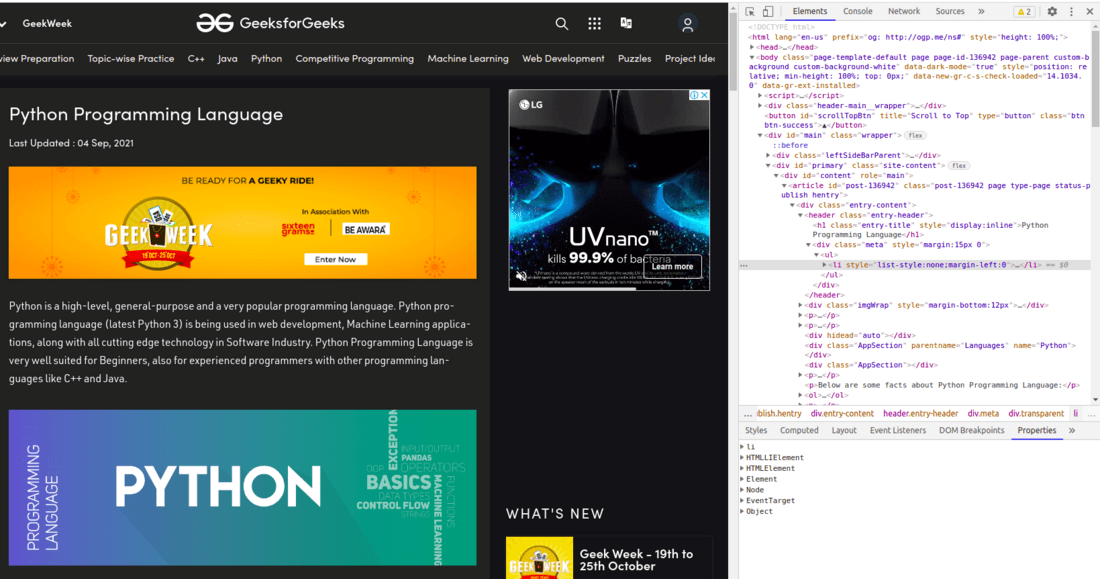


**Inspecting Website**

Before getting out any information from the HTML of the page, we must understand the structure of the page. This is needed to be done in order to select the desired data from the entire page. We can do this by right-clicking on the page we want to scrape and select inspect element.



After clicking the inspect button the Developer Tools of the browser gets open. Now almost all the browsers come with the developers tools installed, and we will be using Chrome for this tutorial.



The developer’s tools allow seeing the site’s [Document Object Model (DOM)](https://www.geeksforgeeks.org/dom-document-object-model/). If you don’t know about DOM, then don’t worry just consider the text displayed as the HTML structure of the page.

**Parsing the HTML**

After getting the HTML of the page let’s see how to parse this raw HTML code into some useful information. First of all, we will create a BeautifulSoup object by specifying the parser we want to use.

**Note:** BeautifulSoup library is built on top of the HTML parsing libraries like html5lib, lxml, html.parser, etc. So BeautifulSoup object and specify the parser library can be created at the same time.

**Example: Python BeautifulSoup Parsing HTML**

**import** requests

**from** bs4 **import** BeautifulSoup

# Making a GET request

r **=** requests.get('<https://www.geeksforgeeks.org/python-programming-language/>')

# check status code for response received

# success code - 200

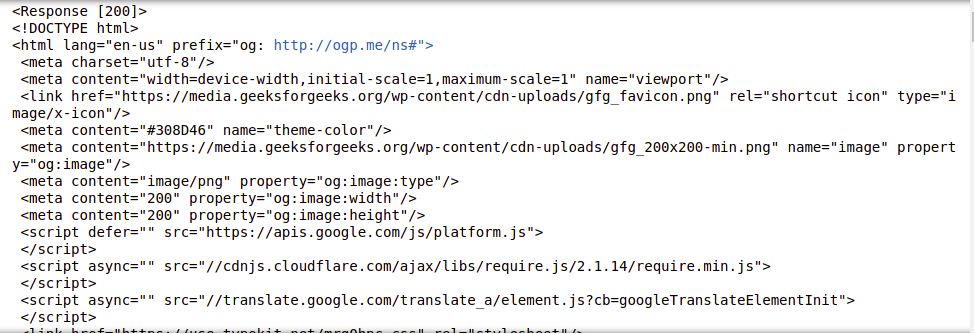
print(r)

# Parsing the HTML

soup **=** BeautifulSoup(r.content, 'html.parser')

print(soup.prettify())

**Output:**



This information is still not useful to us, let’s see another example to make some clear picture from this. Let’s try to extract the title of the page.

**import** requests

**from** bs4 **import** BeautifulSoup

# Making a GET request

r **=** requests.get('<https://www.geeksforgeeks.org/python-programming-language/>')

# Parsing the HTML

soup **=** BeautifulSoup(r.content, 'html.parser')

# Getting the title tag

print(soup.title)

# Getting the name of the tag

print(soup.title.name)

# Getting the name of parent tag

print(soup.title.parent.name)

# use the child attribute to get

# the name of the child tag

**Output:**

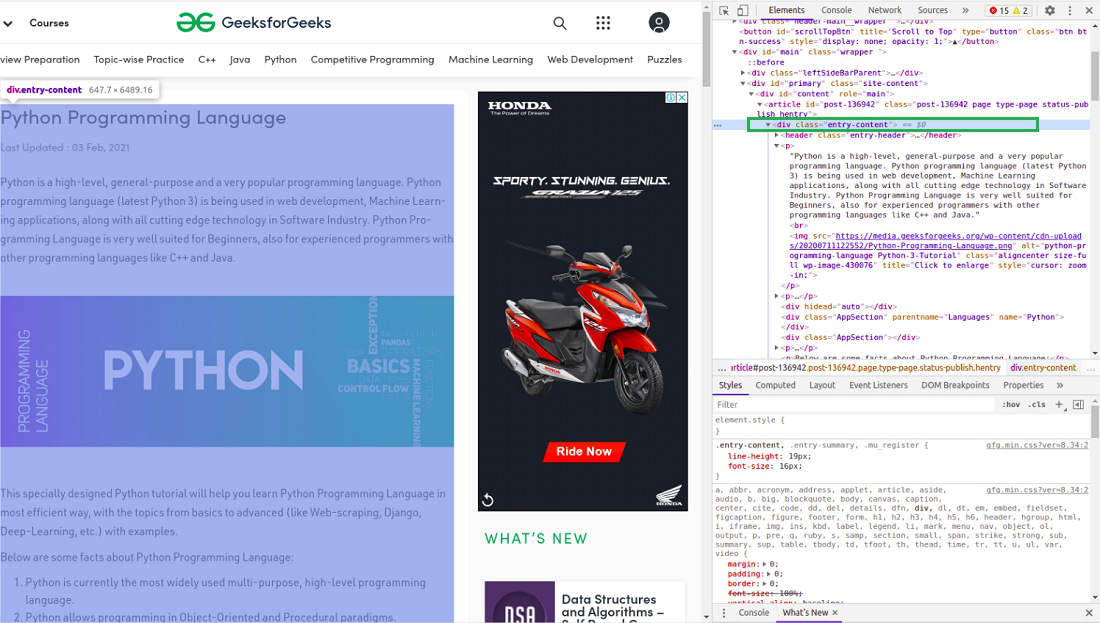
<title>Python Programming Language - GeeksforGeeks</title>

title

html

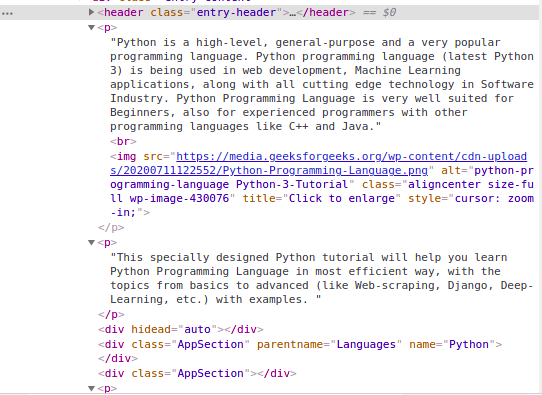
**Finding the Elements**

Now, we would like to extract some useful data from the HTML content. The soup object contains all the data in the nested structure which could be programmatically extracted. The website we want to scrape contains a lot of text so now let’s scrape all those contents. First, let’s inspect the webpage we want to scrape.



**Finding Elements by class**

In the above image, we can see that all the content of the page is under the div with class entry-content. We will use the find class. This class will find the given tag with the given attribute. In our case, it will find all the div having class as entry-content. We have got all the content from the site but you can see that all the images and links are also scraped. So our next task is to find only the content from the above-parsed HTML. On again inspecting the HTML of our website –



We can see that the content of the page is under the <p> tag. Now we have to find all the p tags present in this class. We can use the find\_all class of the BeautifulSoup.

**import** requests

**from** bs4 **import** BeautifulSoup

# Making a GET request

r **=** requests.get('<https://www.geeksforgeeks.org/python-programming-language/>')

# Parsing the HTML

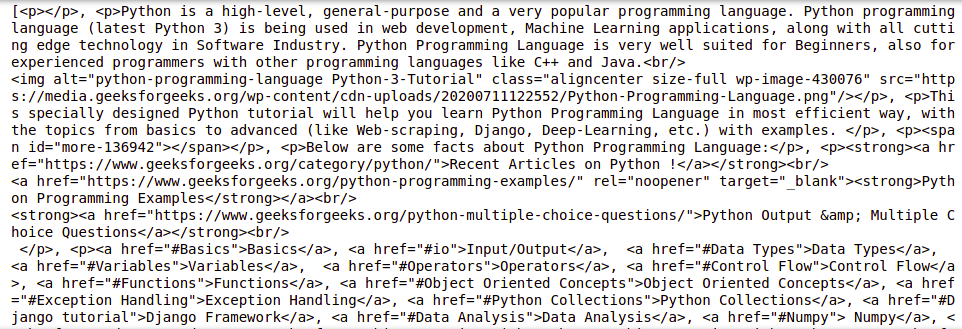
soup **=** BeautifulSoup(r.content, 'html.parser')

s **=** soup.find('div', class\_**=**'entry-content')

content **=** s.find\_all('p')

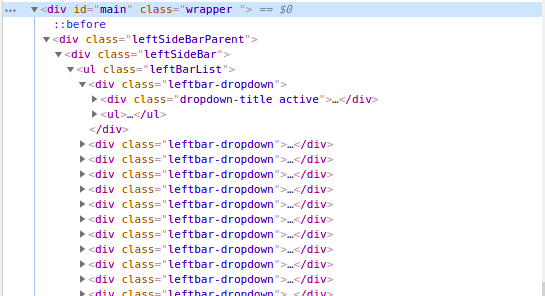
print(content)

**Output:**



**Finding Elements by ID**

In the above example, we have found the elements by the class name but let’s see how to find elements by id. Now for this task let’s scrape the content of the leftbar of the page. The first step is to inspect the page and see the leftbar falls under which tag.



The above image shows that the leftbar falls under the <div> tag with id as main. Now lets’s get the HTML content under this tag. Now let’s inspect more of the page get the content of the leftbar.



We can see that the list in the leftbar is under the <ul> tag with the class as leftBarList and our task is to find all the li under this ul.

**import** requests

**from** bs4 **import** BeautifulSoup

# Making a GET request

r **=** requests.get('<https://www.geeksforgeeks.org/python-programming-language/>')

# Parsing the HTML

soup **=** BeautifulSoup(r.content, 'html.parser')

# Finding by id

s **=** soup.find('div', id**=** 'main')

# Getting the leftbar

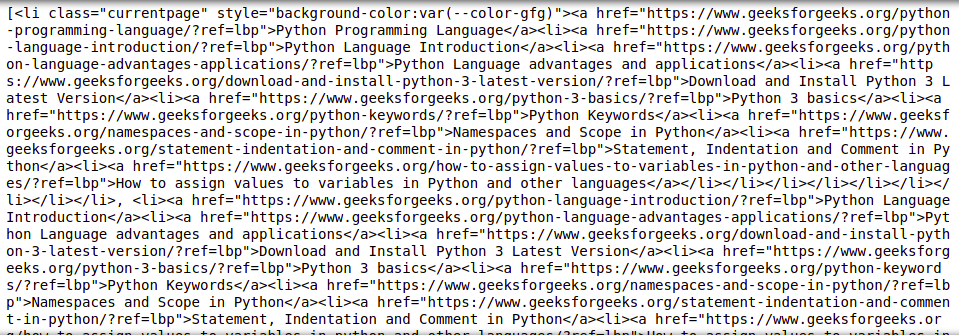
leftbar **=** s.find('ul', class\_**=**'leftBarList')

# All the li under the above ul

content **=** leftbar.find\_all('li')

print(content)

**Output:**



**Extracting Text from the tags**

In the above examples, you must have seen that while scraping the data the tags also get scraped but what if we want only the text without any tags. Don’t worry we will discuss the same in this section. We will be using the text property. It only prints the text from the tag. We will be using the above example and will remove all the tags from them.

**Example 1:** Removing the tags from the content of the page

**import** requests

**from** bs4 **import** BeautifulSoup

# Making a GET request

r **=** requests.get('<https://www.geeksforgeeks.org/python-programming-language/>')

# Parsing the HTML

soup **=** BeautifulSoup(r.content, 'html.parser')

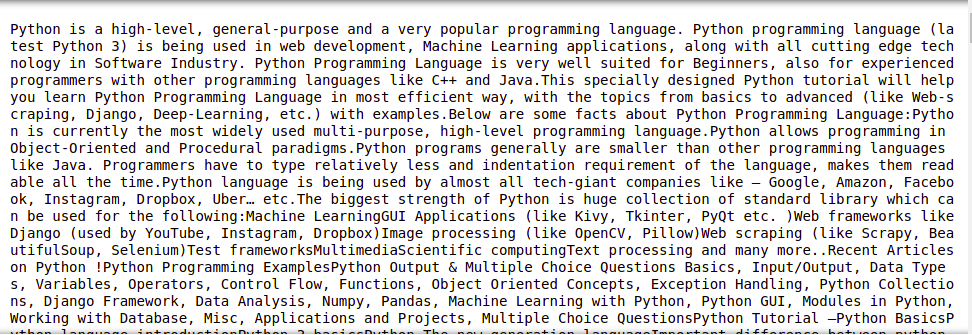
s **=** soup.find('div', class\_**=**'entry-content')

lines **=** s.find\_all('p')

**for** line **in** lines:

**print**(line.text)

**Output:**



Example 2: Removing the tags from the content of the leftbar

**import** requests

**from** bs4 **import** BeautifulSoup

# Making a GET request

r **=** requests.get('<https://www.geeksforgeeks.org/python-programming-language/>')

# Parsing the HTML

soup **=** BeautifulSoup(r.content, 'html.parser')

# Finding by id

s **=** soup.find('div', id**=** 'main')

# Getting the leftbar

leftbar **=** s.find('ul', class\_**=**'leftBarList')

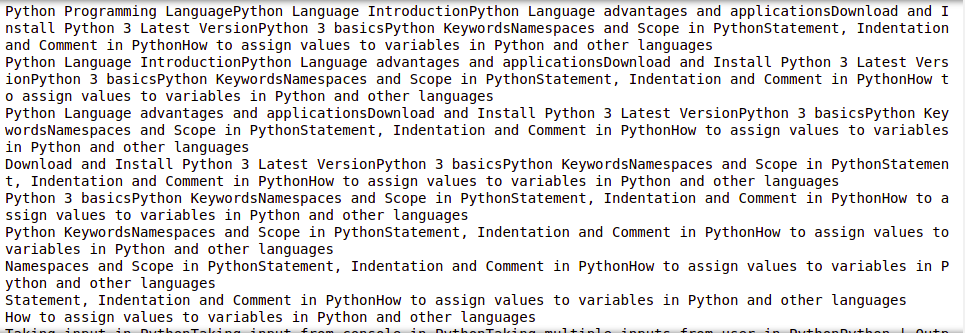
# All the li under the above ul

lines **=** leftbar.find\_all('li')

**for** line **in** lines:

    print(line.text)

**Output:**



**Extracting Links**

Till now we have seen how to extract text, let’s now see how to extract the links from the page.

Example: Python BeautifulSoup Extracting Links

**import** requests

**from** bs4 **import** BeautifulSoup

# Making a GET request

r **=** requests.get('<https://www.geeksforgeeks.org/python-programming-language/>')

# Parsing the HTML

soup **=** BeautifulSoup(r.content, 'html.parser')

# find all the anchor tags with "href"

**for** link **in** soup.find\_all('a'):

    print(link.get('href'))

**Output:**



**Extracting Image Information**

On again inspecting the page, we can see that images lie inside the img tag and the link of that image is inside the src attribute. See the below image –



Example: Python BeautifulSoup Extract Image

**import** requests

**from** bs4 **import** BeautifulSoup

# Making a GET request

r **=** requests.get('<https://www.geeksforgeeks.org/python-programming-language/>')

# Parsing the HTML

soup **=** BeautifulSoup(r.content, 'html.parser')

images\_list **=** []

images **=** soup.select('img')

**for** image **in** images:

    src **=** image.get('src')

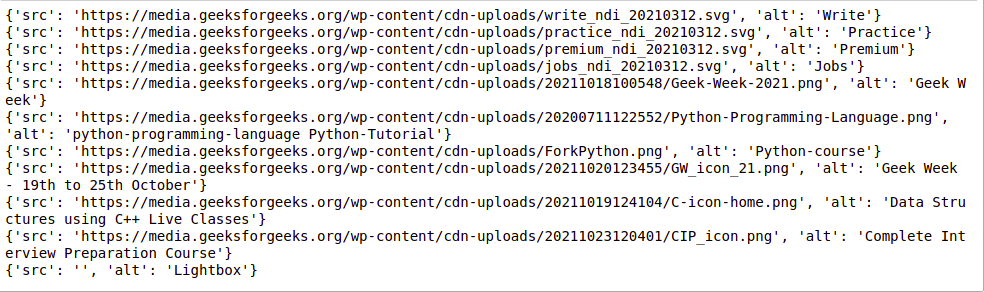
    alt **=** image.get('alt')

    images\_list.append({"src": src, "alt": alt})

**for** image **in** images\_list:

**print**(image)

**Output:**



**Scraping multiple Pages**

Now, there may arise various instances where you may want to get data from multiple pages from the same website or multiple different URLs as well, and manually writing code for each webpage is a time-consuming and tedious task. Plus, it defines all basic principles of automation. Duh!

To solve this exact problem, we will see two main techniques that will help us extract data from multiple webpages:

**Example 1: Looping through the page numbers**



*page numbers at the bottom of the GeeksforGeeks website*

Most websites have pages labeled from 1 to N. This makes it really simple for us to loop through these pages and extract data from them as these pages have similar structures. For example:



*page numbers at the bottom of the GeeksforGeeks website*

Here, we can see the page details at the end of the URL. Using this information we can easily create a for loop iterating over as many pages as we want (by putting page/(i)/ in the URL string and iterating “i” till N) and scrape all the useful data from them. The following code will give you more clarity over how to scrape data by using a For Loop in Python.

**import** requests

**from** bs4 **import** BeautifulSoup as bs

URL **=** '<https://www.geeksforgeeks.org/page/1/>'

req **=** requests.get(URL)

soup **=** bs(req.text, 'html.parser')

titles **=** soup.find\_all('div',attrs **=** {'class','head'})

**print**(titles[4].text)

**Output:**

7 Most Common Time Wastes During Software Development

Now, using the above code, we can get the titles of all the articles by just sandwiching those lines with a loop.

**import** requests

**from** bs4 **import** BeautifulSoup as bs

URL **=** '<https://www.geeksforgeeks.org/page/>'

**for** page **in** range(1, 10):

    req **=** requests.get(URL **+** str(page) **+** '/')

    soup **=** bs(req.text, 'html.parser')

    titles **=** soup.find\_all('div', attrs**=**{'class', 'head'})

**for** i **in** range(4, 19):

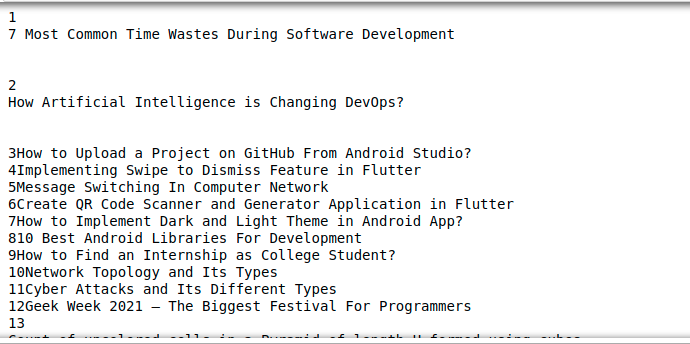
**if** page > 1:

**print**(f"{(i-3)+page\*15}" **+** titles[i].text)

**else**:

**print**(f"{i-3}" **+** titles[i].text)

**Output:**



**Example 2: Looping through a list of different URLs**

The above technique is absolutely wonderful, but what if you need to scrape different pages, and you don’t know their page numbers? You’ll need to scrape those different URLs one by one and manually code a script for every such webpage.

Instead, you could just make a list of these URLs and loop through them. By simply iterating the items in the list i.e. the URLs, we will be able to extract the titles of those pages without having to write code for each page. Here’s an example code of how you can do it.

**import** requests

**from** bs4 **import** BeautifulSoup as bs

URL **=** ['[https://www.geeksforgeeks.org](https://www.geeksforgeeks.org/)','<https://www.geeksforgeeks.org/page/10/>']

**for** url **in** range(0,2):

    req **=** requests.get(URL[url])

    soup **=** bs(req.text, 'html.parser')

    titles **=** soup.find\_all('div',attrs**=**{'class','head'})

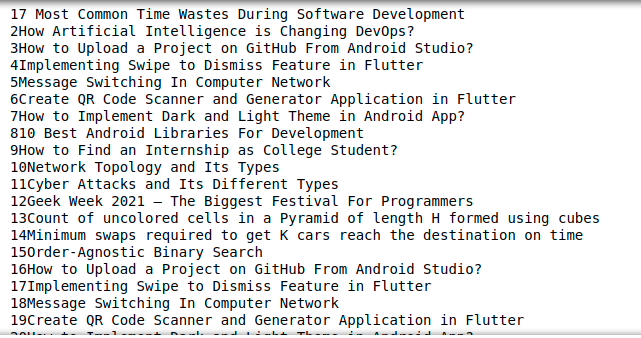
**for** i **in** range(4, 19):

**if** url**+**1 > 1:

**print**(f"{(i - 3) + url \* 15}" **+** titles[i].text)

**else**:

**print**(f"{i - 3}" **+** titles[i].text)

**Output:**

**Saving Data to CSV**

First we will create a list of dictionaries with the key value pairs that we want to add in the CSV file. Then we will use the csv module to write the output in the CSV file. See the below example for better understanding.

**Example: Python BeautifulSoup saving to CSV**

**import** requests

**from** bs4 **import** BeautifulSoup as bs

**import** csv

URL **=** '<https://www.geeksforgeeks.org/page/>'

soup **=** bs(req.text, 'html.parser')

titles **=** soup.find\_all('div', attrs**=**{'class', 'head'})

titles\_list **=** []

count **=** 1

**for** title **in** titles:

    d **=** {}

    d['Title Number'] **=** f'Title {count}'

    d['Title Name'] **=** title.text

    count **+=** 1

    titles\_list.append(d)

filename **=** 'titles.csv'

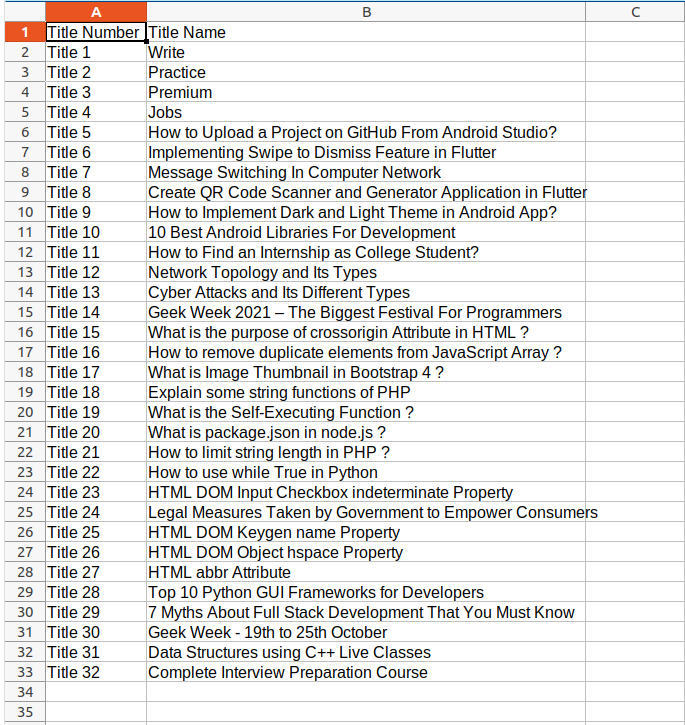
with open(filename, 'w', newline**=**'') as f:

    w **=** csv.DictWriter(f,['Title Number','Title Name'])

    w.writeheader()

    w.writerows(titles\_list)

**Output:**



**Python Libraries**

There are several Python libraries that can be used to build a web scraper, such as BeautifulSoup, Scrapy, and Selenium.

BeautifulSoup is a library for pulling data out of HTML and XML files. It can be used to extract specific elements from a webpage, such as links, images, and text.

Scrapy is an open-source and collaborative web crawling framework for Python. It is used to extract the data from websites by following links and saving the data in structured format.

Selenium is a library that allows you to automate web browsers. It can be used to automate tasks such as filling out forms, clicking buttons, and navigating pages. This can be useful for scraping websites that use JavaScript to load content.

**Note:**

When building a web scraper, it is important to be aware of the terms of service of the website you are scraping and to respect the website's request rate limits. Some websites may also block scraper IP addresses.

Web scraping can be a powerful tool for extracting data from the web, but it should be used responsibly and with respect for website owners' terms of service.

**Misconceptions About Web Scraping**

1. **Web scraping is illegal:** While web scraping can be used for illegal activities, such as scraping personal data or copyrighted content, web scraping itself is not illegal. However, the terms of service of a website may prohibit web scraping, and some websites may have security measures in place to block scrapers. It's important to check a website's terms of service and to respect any requests from website owners to stop scraping.
2. **Web scraping is easy:** Web scraping can be a complex task, especially for websites that use technologies such as JavaScript to load content, or for websites that have a large amount of data. It can take a significant amount of time and effort to build a web scraper that can effectively extract the desired data from a website.
3. **Web scraping is only for big companies:** Web scraping can be useful for a wide range of tasks, and it is not limited to large companies. Small businesses, researchers, and individuals can also use web scraping to gather data for a variety of purposes, such as market research, price comparison, or data analysis.
4. **Web scraping is only for extracting text**: Web scraping can be used to extract a wide range of data, not just text. It can be used to extract images, videos, and other multimedia content, as well as structured data such as prices and product specifications.
5. **Web scraping is only done with Python:** While Python is a popular language for web scraping, it is not the only one. Other programming languages such as Java, C# and Ruby also have libraries and frameworks that can be used for web scraping.

**Getting Started Scraping**

Here are the general steps to get started with web scraping:

1. **Choose a website to scrape:** Select a website that has the data you need and that allows web scraping in its terms of service.
2. **Inspect the website:** Use a web browser's developer tools to inspect the website's HTML code and understand the structure of the data you want to scrape.
3. **Choose a scraping library:** Select a scraping library or framework, such as BeautifulSoup, Scrapy or Selenium, that is compatible with the website and the data you want to scrape.
4. **Write the code:** Using the library or framework you've chosen, write the code to extract the data you want. This typically involves making HTTP requests to the website, parsing the HTML or XML response, and extracting the desired data.
5. **Test and debug your code:** Test your code by running it and check the output to make sure it's extracting the data correctly. Debug any errors or issues that arise.
6. **Extract and store the data:** Once you have your code working correctly, you can extract the data from the website and store it in a file or a database.
7. **Optimize and schedule:** Consider setting up your scraping script to run on schedule and also optimize the script for performance, such as by implementing pagination, limiting requests rate and handling errors.

It's important to note that web scraping can be a complex task, and it may take some time to learn the necessary skills and to build a functional web scraper. It is also important to consider the legal and ethical implications of web scraping and to always respect a website's terms of service.