Introduction to Python

Fondren Library
Research Data Services


Web Scraping
BeautifulSoup +
Requests

Web Scraping and Data Cleaning

What is web scraping?

Web scraping is the **automated process of extracting data from websites**, transforming vast amounts of online information into accessible, manipulatable, and machine-readable formats. This technique allows users to turn **complex websites into concise summaries**, convert blogs into detailed biographies, and make any web-based data easily accessible for analysis. By automating data collection, web scraping enables us to **efficiently gather and process information** that would otherwise require extensive manual effort. It opens up a world of possibilities for research, content creation, and data-driven decision-making. It opens up the question -- **what websites would you like to scrape?**

How Web Scraping Works

Web scraping automates the process of extracting data from websites by sending requests to web pages, parsing the HTML, and retrieving specific information. This involves using scripts to navigate through the web content, identify the required data elements, and systematically extract them. The collected data is then stored in various formats such as databases or spreadsheets for further analysis.

Web Scraping breakdown:

- **Automated Requests:** Scripts send requests to web pages to retrieve content.
- **HTML Parsing:** Tools parse the HTML structure to locate specific data elements.
- **Data Extraction:** Desired information is systematically extracted from the web page.
- **Data Storage:** Extracted data is saved in formats like databases, CSV files, or spreadsheets.

What’s inside a website

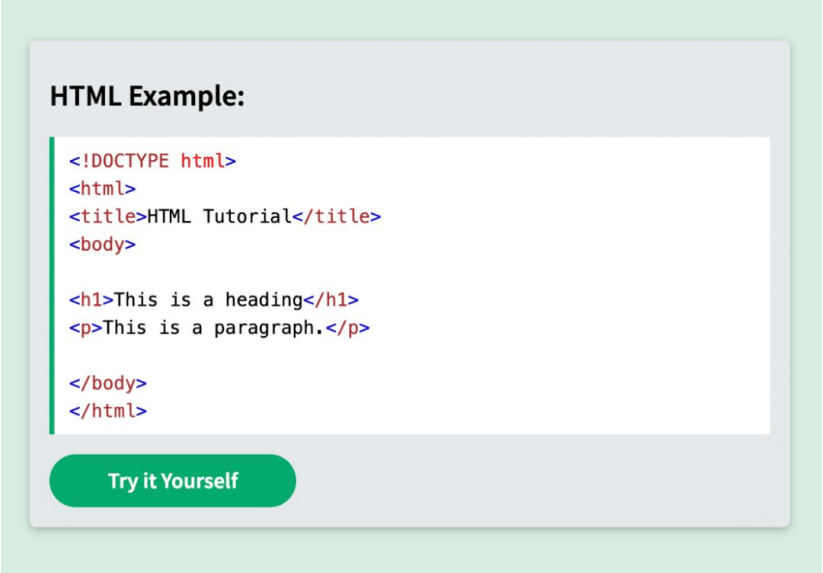
HTML (HyperText Markup Language) structures the content on web pages using elements defined by tags. Understanding HTML is crucial for web scraping as it allows you to navigate and extract the required data efficiently.

HTML Tags:

HTML tags are the building blocks of web pages, defining the structure and content elements within a webpage.

- <html>: Defines the root of an HTML document.
- <head>: Contains meta-information about the document.
- <title>: Sets the title of the web page, shown in the browser's title bar.
- <body>: Encloses the main content of the web page.
- <h1> to <h6>: Define headings, <h1> being the highest level and <h6> the lowest.
- <p>: Defines a paragraph block of text.
- <a>: Creates a hyperlink to another page or resource.
- <img>: Embeds an image in the web page.
- : Defines a division or section, used to group elements.

By understanding HTML tags, you’ll be able to more easily identify which parts of a page’s code contain the information you’re most interested in scraping!



Types of Data on the Web

Web data comes in various forms, each with unique structures and characteristics. Understanding the different types of data is crucial for effective web scraping and data analysis.

Structured Data:

- Organized in a fixed format, like tables.
- Easy to parse and extract.
- Example: HTML tables, databases.

```html
<table>
  <tr>
    <th>Name</th>
    <th>Age</th>
  </tr>
  <tr>
    <td>John Doe</td>
    <td>30</td>
  </tr>
</table>
```

Semi-structured Data:

- Not organized in a fixed schema, but still contains tags or markers.
- Requires parsing to extract meaningful information.
- Example: JSON, XML.

```json
{
  "name": "John Doe",
  "age": 30
}
```

```xml
<person>
  <name>John Doe</name>
  <age>30</age>
</person>
```

Unstructured Data:

- No predefined structure.
- Requires significant processing to extract useful information.
- Example: Plain text, articles, blog posts.

```html
<div>
  <p>John Doe is a software engineer.</p>
  <p>He started his career in a small startup.</p>
  <p>John has worked on various projects.</p>
  <p>He is currently focused on AI solutions.</p>
</div>
```

Data vs Metadata

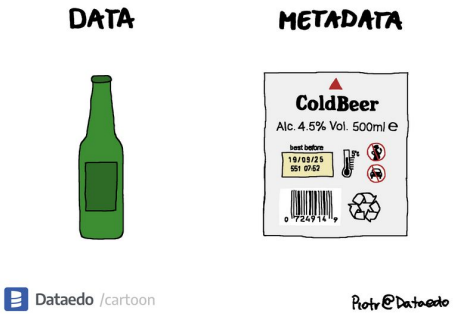
Understanding the distinction between data and metadata is foundational for web scraping and data management. Data represents the core information we seek to analyze, while metadata provides essential context that enhances the usability, discoverability, and understanding of that data. This distinction is critical for organizing and making sense of large datasets.

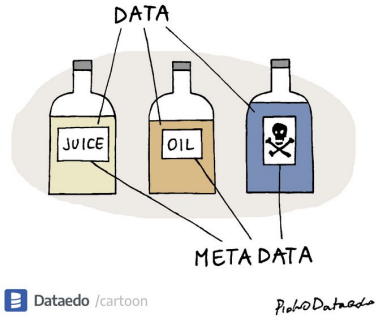
Data:

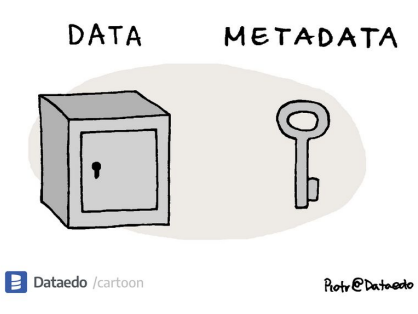
- **Definition:** The actual content or information collected.
- **Example:** The text of a blog post, numerical values in a table.
- **Usage:** Directly used for analysis, creating reports, generating insights, and making decisions.
- **Significance:** Without data, there is nothing to analyze or interpret

Metadata:

- **Definition:** Descriptive information about the data.
- **Example:** Author of the blog post, date published, keywords, file size, data types.
- **Usage:** Facilitates data organization, improves searchability, enhances data management, and provides context.
- **Significance:** Without metadata, understanding the origin, relevance, and structure of the data becomes challenging, leading to potential misinterpretations.







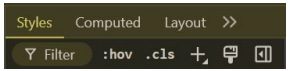
Using Inspect Element

The Inspect Element tool in web browsers allows you to explore and interact with the HTML and CSS of a webpage. This tool is invaluable for web scraping as it helps you identify and isolate the elements containing the data you need.

Identifying elements:

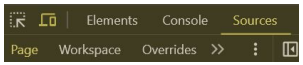
To open inspect element, right-click on a web page and select "Inspect" or press `Ctrl+Shift+I` (Windows) / `Cmd+Opt+I` (Mac).This opens the Developer Tools panel, showing the HTML and CSS of the page. Use the Inspect tool to hover over elements on the webpage. The corresponding HTML code will be highlighted in the Developer Tools panel.

Copying Selectors:

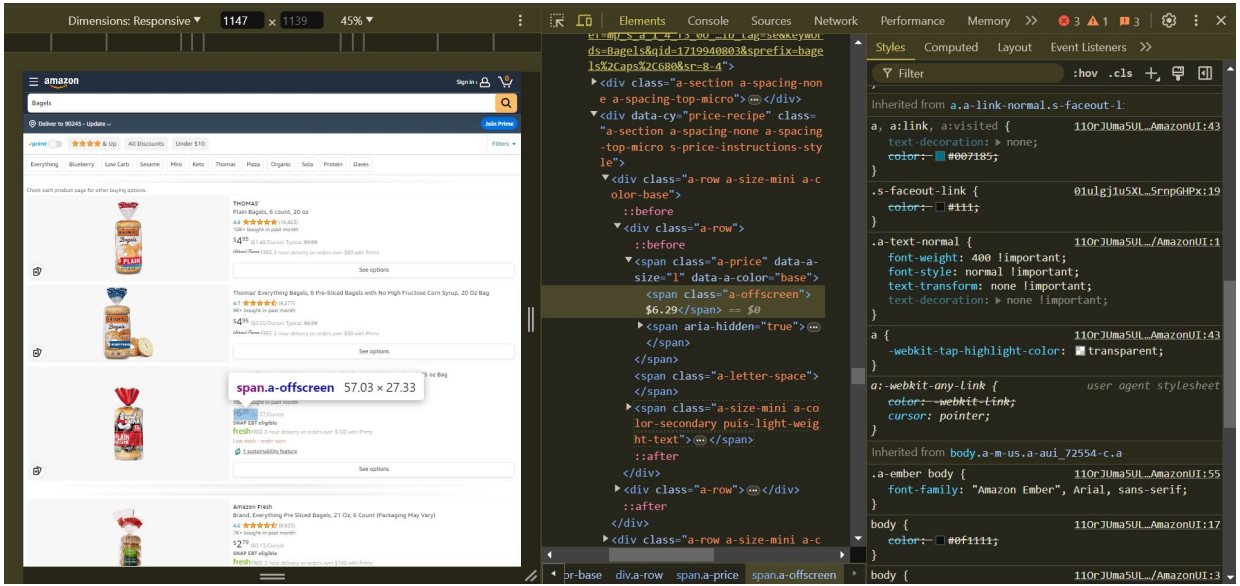


Right-click on the highlighted HTML and select "Copy" -> "Copy selector" to get the CSS selector for the element.

View Source Code:



To view source code information, and see assets such as images, icons, sound files, etc, click on the Sources tab.



What are Web Scrapers

Web scrapers are tools, libraries, or scripts designed to extract data from websites. Using an understanding of HTML tags, and programming logic in the networking domain, they make requests to websites to extract the source code, and then based on the HTML tags, they extract specific elements of the website to return to you in the form of csvs, jsons, or text files.

Network Requests:

- **GET Requests:** Retrieve the content of web pages by sending HTTP GET requests.
- **Usage:** Essential for accessing the raw HTML of a web page

```
import requests
response = requests.get('https://example.com')
```

HTML Parsing:

- **Parsing HTML:** Extract and navigate HTML elements to locate specific data.
- **Usage:** Used to identify and extract data points like headings, paragraphs, and tables.

```
soup = BeautifulSoup(response.content, 'html.parser')

# Find all <h1> tags
h1_tags = soup.find_all('h1')
for tag in h1_tags:
    print(tag.text)
```

Handling Dynamic Content:

- **JavaScript Rendering:** Manage content loaded dynamically via JavaScript.
- **Usage:** Necessary for scraping sites where content is not present in the initial HTML.

```
driver = webdriver.Chrome()
driver.get('https://example.com')

# Click a link to navigate
link = driver.find_element(By.LINK_TEXT, 'Next Page')
link.click()
```

Authentication and Sessions:

- **Login Handling:** Manage sites that require user authentication.
- **Usage:** Needed for accessing data behind login pages or maintaining sessions.

```
payload = {'username': 'user', 'password': 'pass'}
with requests.Session() as session:
  post = session.post('https://example.com/login', data=payload)
```

API Interaction:

- **Using APIs:** Interact with web APIs to fetch structured data directly.
- **Usage:** Efficient for retrieving data without parsing HTML, often faster and more reliable.

```
response = requests.get('https://api.example.com/data')
data = response.json()
```

Data Storage:

- **Storing Data:** Save extracted data into formats like CSV, JSON, or databases.
- **Usage:** Ensures data is organized and ready for analysis.

Beautiful Soup

BeautifulSoup is a powerful Python library for parsing HTML and XML documents. It creates a parse tree from the page's source code, allowing for easy navigation and data extraction.

Install BeautifulSoup and Requests:

- Use pip to install the necessary libraries.
  ```
  pip install beautifulsoup4 requests
  ```

Import Libraries:

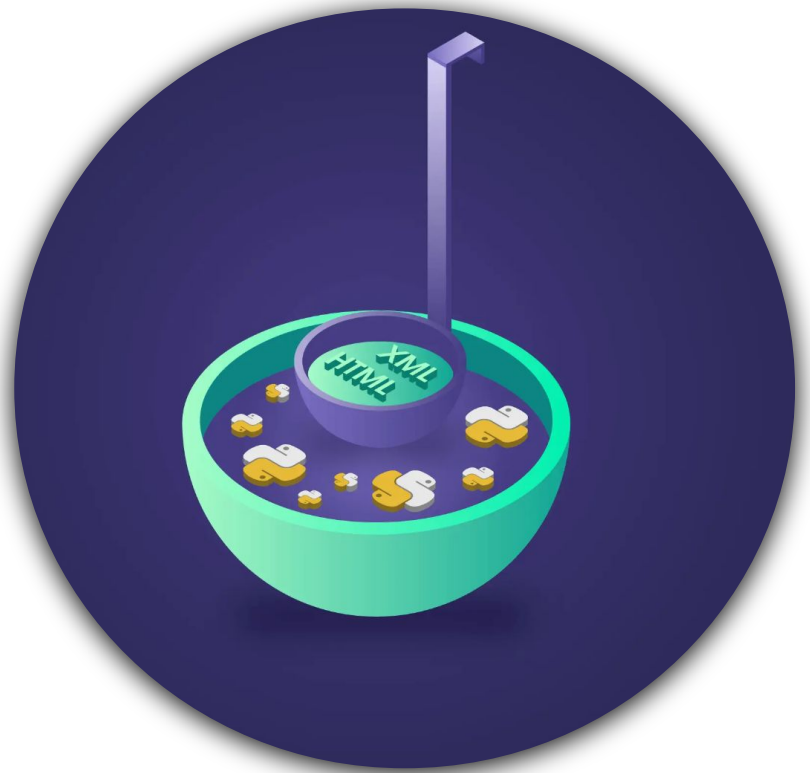
- Import BeautifulSoup and requests in your Python script.
  ```
  from bs4 import BeautifulSoup
  import requests
  ```

Send a GET Request to the Website:

- Use the requests library to fetch the webpage content.
  ```
  url = 'https://example.com
  response = requests.get(url)
  ```

Parse the HTML Content:

- Create a BeautifulSoup object to parse the HTML.
  ```
  soup = BeautifulSoup(response.content, 'html.parser')
  ```



Beautiful Soup - Navigating the Parse Tree

The parse tree is a hierarchical representation of an HTML document created by BeautifulSoup. It reflects the nested structure of HTML elements, allowing users to navigate and manipulate the document easily. By traversing the parse tree, users can access parent, child, and sibling elements, making it straightforward to locate and extract specific data from the HTML.

Navigating the parse tree:

Find Elements by Tag:

- Use BeautifulSoup to find elements by their HTML tags.

```
h1_tag = soup.find('h1')
print(h1_tag.text)
```

Find Elements with Attributes:

- Find elements using their attributes, such as class or id.

```
special_divs = soup.find_all('div', class_='special')
for div in special_divs:
print(div.text)
```

Find Elements by Text:

- Locate elements that contain specific text.

```
elements = soup.find_all(text='Specific Text')
for element in elements:
print(element)
```

Find elements by relation:

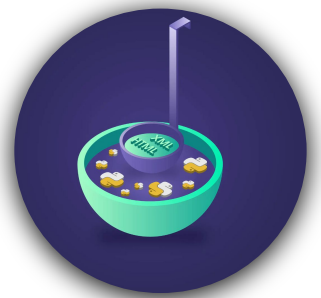
- Access parent and child elements to navigate the HTML structure.

```
parent = h1_tag.parent
children = parent.find_all('p')
for child in children:
print(child.text)
```

Extract Links:

- Extract and print all hyperlinks (href attributes) on the page.

```
links = soup.find_all('a')
for link in links:
print(link.get('href'))
```



Cleaning Unstructured Data

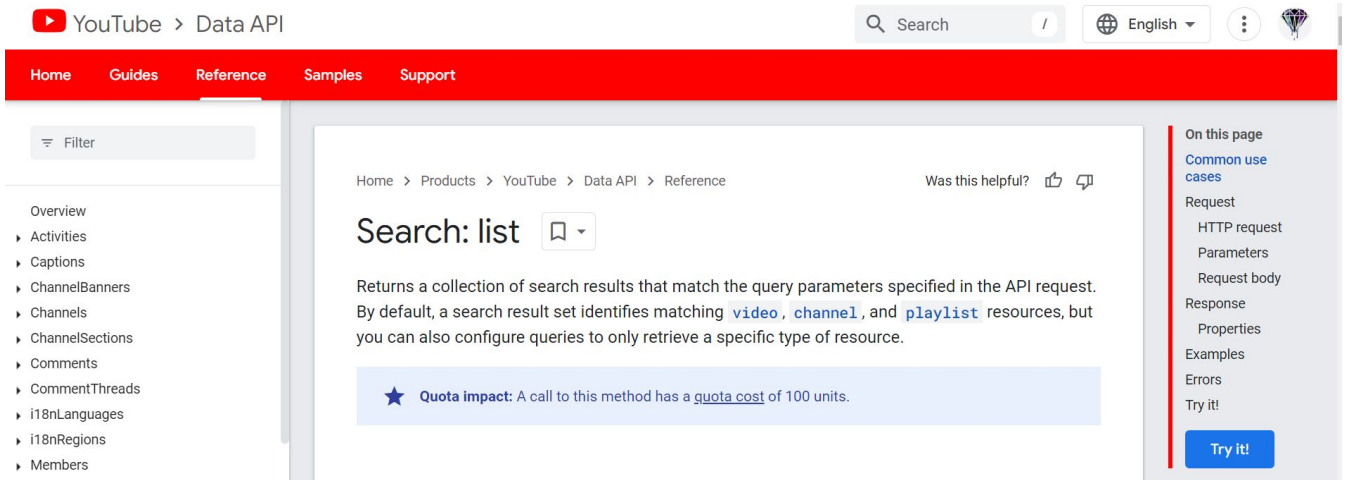
Unstructured data, such as text from articles, social media posts, or customer reviews, is often messy and difficult to analyze in its raw form. Cleaning this data is crucial to make it suitable for analysis. The first step is text parsing, which involves extracting relevant information from the text. For example, using regular expressions to find email addresses or phone numbers within a body of text. Removing noise is another key step, which includes eliminating unnecessary elements like HTML tags, punctuation, and stop words. For instance, stripping HTML tags using BeautifulSoup can clean up text extracted from web pages, making it more readable and easier to process.

Tokenization is the process of breaking down text into individual words or tokens, which is essential for text analysis. For example, splitting the sentence "This is a sample sentence." into ["This", "is", "a", "sample", "sentence"]. Normalization involves standardizing the text to a consistent format, such as converting all text to lowercase to avoid case sensitivity issues. Additionally, stemming and lemmatization reduce words to their root forms, so that "running," "ran," and "runs" are all treated as the base word "run." These steps help in reducing the complexity of the data and improving the accuracy of subsequent analysis, allowing for more meaningful insights to be derived from the cleaned data.

API Scraping

APIs (Application Programming Interfaces) offer a more efficient and reliable way to gather data compared to traditional web scraping methods. They provide structured access to a website's data, allowing for direct requests and responses without the need to parse HTML. APIs return data in structured formats like JSON or XML, which simplifies data extraction and processing. For instance, using a weather API to get current weather information directly in a machine-readable format is far more efficient than scraping the weather website's HTML.

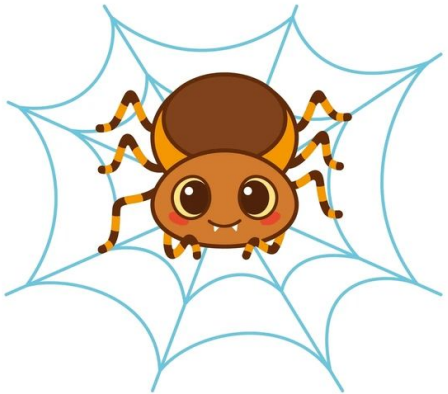
However, not all websites, especially smaller ones, offer APIs. To check if a site has an API, you can look for an API documentation link on the website, often found in the footer or under the developer section. Additionally, you can perform a web search for "[website name] API" to find any available documentation or developer resources. If an API is available, it will typically include information on how to authenticate, make requests, and handle responses. Using APIs where available is a best practice for web scraping, as it is more reliable and respectful of the website's resources.



Scraping Safely - Best Practices

As exciting as web scraping is, when done irresponsibly, it can have significant consequences. Overloading websites with frequent and aggressive scraping can harm their performance, disrupt user experience, and even lead to legal repercussions. By scraping responsibly, we show respect for the hard work of web developers and maintain the stability and accessibility of online services for everyone. Ethical web scraping ensures that we can continue to gather the data we need without causing harm to the digital ecosystem.

1. **Respect Website Terms of Service or Privacy Policy/Agreement:**
    - Always review and adhere to the website's terms of service to ensure your scraping activities are legal and ethical.
    - Avoid scraping websites that explicitly prohibit it. Public, Public-Facing, Open Source are all words that tell us that a website is open or accessible to web scraping tools.
2. **Rate Limiting:**
    - Implement rate limiting to avoid overwhelming the target website with requests.
    - Introduce delays between requests to mimic human browsing behavior.
3. **Handling HTTP Errors:**
    - Check for HTTP status codes and handle errors gracefully.
    - If a request fails, ensure your scraper can handle the error without crashing.
4. **Respect Robots.txt:**
    - Check and respect the website's `robots.txt` file to see which parts of the site can be crawled.
    - Follow the guidelines provided by the website to avoid any potential conflicts.
5. **Monitoring and Logging:**
    - Keep records of requests and responses to debug and improve your scraping strategy.
6. **Polite Scraping:**
    - Avoid scraping during peak hours to minimize impact on the website's performance.
    - Identify yourself by setting a user-agent string that includes your contact information.



If it is vague whether or not you can scrape a website, try to contact the developers, administrators, or tech support to formally request permission!

Common Challenges

1. **Dynamic Content:**
   - **Challenge:** Websites that load content dynamically via JavaScript can be difficult to scrape.
   - **Solution:** Use tools like Selenium to render the page and capture the fully loaded content.
2. **Anti-Scraping Measures:**
   - **Challenge:** Websites may implement anti-scraping measures such as CAPTCHA, IP blocking, or user-agent detection.
   - **Solution:** Respect these measures and consider reaching out for API access or official data sharing agreements.
3. **Frequent Website Changes:**
   - **Challenge:** Websites frequently update their structure, breaking scraping scripts.
   - **Solution:** Regularly update your scraping scripts and use flexible parsing methods to adapt to changes.
4. **Legal and Ethical Concerns:**
   - **Challenge:** Navigating the legal and ethical landscape of web scraping can be complex.
   - **Solution:** Always adhere to the website's terms of service, respect `robots.txt`, and consider the ethical implications of your scraping activities.
5. **Data Quality:**
   - **Challenge:** Extracted data may be incomplete, inconsistent, or noisy.
   - **Solution:** Implement thorough data cleaning and validation processes to ensure accuracy and usability.
6. **Performance and Scalability:**
   - **Challenge:** Scraping large volumes of data can be resource-intensive and slow.
   - **Solution:** Optimize your scraping scripts for performance and use distributed systems to scale the scraping process.

Web Scrape Showcase

Many incredible things have been made using the data gathered through web scraping. When done responsibly and effectively, you can make exciting applications, research projects, compilations of data, and even AI models with web scraping. Here are some examples of how web scraped data has contributed to amazing applications.

1. **ChatGPT and AI Models:**
   - Large language models like ChatGPT are trained on vast amounts of text data from the web. This data is web-scraped from diverse sources to provide a rich dataset for training.
2. **Market Research and Analysis:**
   - Companies scrape data from e-commerce sites, social media, and review platforms to analyze market trends, customer sentiment, and competitor pricing.
3. **Real Estate and Property Insights:**
   - Websites like Zillow and Redfin scrape data from various sources to provide comprehensive real estate listings, price trends, and property analytics.
4. **Financial Analytics and Stock Market Predictions:**
   - Financial firms scrape data from news sites, financial reports, and social media to predict stock market trends and perform risk analysis.
5. **Academic Research:**
   - Researchers scrape data from academic journals, public datasets, and other online sources to conduct meta-analyses, literature reviews, and empirical studies.
6. **Public Health and Epidemiology:**
   - Health organizations scrape data from online health forums, social media, and news sites to track disease outbreaks and public health trends.

Bloomberg Business

CNN