Data Crawler tools

There are several Python libraries and frameworks that can be used for web scraping and data crawling. Some of the popular tools for data crawling in Python are:

1. **Beautiful Soup**: It is a Python library used for web scraping purposes to pull the data out of HTML and XML files. It provides a simple way to navigate, search, and modify the parse tree.
2. **Scrapy:** It is an open-source and collaborative web crawling framework used to extract the data from websites. Scrapy provides an integrated way for handling requests, managing spiders, and processing data. At Carro, we use Scrapy to crawl data from competitor sites.
3. **Selenium**: It is a browser automation tool used for web scraping and testing purposes. It can simulate user actions, such as clicking buttons and filling out forms, which makes it useful for interacting with dynamic websites.
4. **Requests**: It is a simple HTTP library used for sending HTTP requests in Python. Requests can be used to crawl and extract data from websites, but it requires additional libraries like BeautifulSoup for parsing the HTML content.
5. **PyQuery**: It is a Python library used for web scraping with jQuery syntax. PyQuery allows you to select elements from HTML documents using jQuery-like syntax, making it easy to extract the desired data.

What is Scapy? Scapy example using Python

Scapy is a Python-based packet manipulation tool and library that allows you to craft and decode network packets. It is an open-source library that can be used for network testing, troubleshooting, and security assessment. Scapy provides a user-friendly interface to interact with the underlying network stack and can be used to create custom network tools and protocols. It also provides a high-level interface for crawling websites and extracting structured data from them. Scrapy supports various protocols and technologies, including HTTP, HTTPS, FTP, XML, and JSON.

Here is an example of how to use Scapy in Python to send an ICMP (Internet Control Message Protocol) ping request:

```
1 from scapy.all import *
2
3 # Create an ICMP packet with destination IP address
```

```
packet = IP(dst="www.google.com")/ICMP()

# Send the packet and receive the response
reply = sr1(packet, timeout=2)

# Check if a response was received
if reply is not None:
    # Print the source IP address and the round-trip time
    print("Ping reply from", reply.src, "in", round((reply.time - packet.sent_ti
 else:
     # Print an error message if no response was received
     print("No response received.")

```

In this example, we use Scapy to create an IP packet with a destination address of `www.google.com`, followed by an ICMP packet. We then use the `sr1()` function to send the packet and receive a single response within a timeout of 2 seconds. If a response is received, we print the source IP address and the round-trip time. If no response is received, we print an error message.

Another example, to extract data from Yahoo Finance using Scrapy, we can follow these steps:

1. Create a new Scrapy project by running the following command in the terminal:

```
scrapy startproject yahoo_finance

```

This will create a new directory called "yahoo_finance" with the basic structure of a Scrapy project.

2. Create a new Spider by running the following command in the terminal:

```
scrapy genspider yahoo_spider finance.yahoo.com
```

This will create a new Python file called "yahoo_spider.py" inside the "spiders" directory. We can edit this file to define the rules for crawling Yahoo Finance.

3. Edit the "yahoo_spider.py" file to add the rules for crawling Yahoo Finance. For example, to extract the titles and links of the news articles from the "Top News" section, and extract data for each stock quote listed in the "Market Summary" section. We can add the following code:

```
# yahoo_spider.py
import scrapy

class YahooSpider(scrapy.Spider):
    name = 'yahoo'
    allowed_domains = ['finance.yahoo.com']
    start_urls = ['https://finance.yahoo.com']

    def parse(self, response):
        top_news = response.css('#Lead-1-QuoteController .StreamMegaItem')
        for news in top_news:
            yield {
                'title': news.css('.StreamMegaItem-title::text').get(),
                'link': news.css('.StreamMegaItem-title::attr(href)').get()
            }
        for quote in response.css('ul#market-summary > li'):
            yield {
                'symbol': quote.css('a > span::text').get(),
                'name': quote.css('a > span::attr(tittle)').get(),
                'price': quote.css('span[data-reactid="32"]::text').get(),
            }


class YahooFinanceSpider(scrapy.Spider):
    name = 'yahoo_finance'
    allowed_domains = ['finance.yahoo.com']
    start_urls = ['https://finance.yahoo.com']

    def parse(self, response):
        for quote in response.css('ul#market-summary > li'):
            yield {
                'symbol': quote.css('a > span::text').get(),
                'name': quote.css('a > span::attr(tittle)').get(),
                'price': quote.css('span[data-reactid="32"]::text').get(),
            }

```

4. Run the Spider by running the following command in the terminal:

```

```

```
scrapy crawl yahoo_finance -o quotes.csv
```

This will start the Spider and extract the data from Yahoo Finance according to the rules defined in the "yahoo_spider.py" file. The extracted data will be printed on the terminal in JSON format to the file called 'quotes.csv'.

Note: We may need to install additional Scrapy packages to handle different types of data on the web pages. For example, we may need to install "scrapy-splash" to handle websites that require JavaScript rendering.

Libraries and functions in Scrapy

Scrapy is a Python framework for web scraping and crawling. It provides several libraries and functions to make it easy to extract data from websites. Some of the main libraries and functions in Scrapy include:

1. Selectors: Scrapy uses selectors to extract data from HTML or XML documents. Selectors allow you to specify the elements or attributes you want to extract using CSS or XPath expressions.
2. Item: Items are simple containers used to collect the data extracted from websites. They can be defined as Python classes with attributes that represent the data you want to extract.
3. Spider: Spiders are the main components of Scrapy. They define how to navigate through websites and extract data using selectors. Spiders can follow links to other pages and can be configured to run periodically.
4. Pipeline: Pipelines are used to process the data collected by Scrapy. They can be used to clean, validate, and store the data in various formats such as CSV, JSON, or databases.
5. Settings: Scrapy provides a settings module that allows you to configure various aspects of the framework. You can use settings to configure the user-agent, set download delays, enable or disable cookies, and many other options.
6. Downloader: The downloader is responsible for fetching the web pages specified by the spider. Scrapy provides a powerful downloader that can handle asynchronous requests, follow redirects, and handle cookies.
7. Middleware: Scrapy provides a middleware system that allows you to add functionality to the spider. Middleware can be used to modify requests and responses, add custom headers, handle errors, and more.

Overall, Scrapy is a comprehensive web scraping framework that provides a wide range of libraries and functions to make it easy to extract data from websites.

Example of PyQuery to extract data from a specific web, store into AWS S3, schedule job everyday at 2AM UTC+07

Here is an example of how to use PyQuery to extract data from a specific website and store it in AWS S3, and then schedule the job to run every day at 2AM UTC+07.

This code will extract the data from the specified website using PyQuery, store it in an S3 bucket in AWS, and then schedule the job to run every day at 2AM UTC+07. Just replace the `url`, `bucket_name`, `s3_prefix`, and `my-selector` values with the relevant ones for your use case.

```
import boto3
import pyquery
import requests
from datetime import datetime, timedelta
import pytz

# Define the URL of the website to scrape
url = 'https://example.com'

# Define the S3 bucket to store the scraped data
bucket_name = 'my-bucket'
s3_prefix = 'data/'

# Set the timezone to UTC+07
timezone = pytz.timezone('Asia/Bangkok')

# Set the time to run the job every day at 2AM UTC+07
scheduled_time = datetime.now(timezone).replace(hour=2, minute=0, second=0, micr

# Define a function to scrape the data and store it in S3
def scrape_and_store_data():
    # Use requests to get the HTML content of the webpage
    response = requests.get(url)
    html_content = response.content

    # Use PyQuery to extract the relevant data from the HTML
    pq = pyquery.PyQuery(html_content)
    data = pq('.my-selector').text()

```

```
    # Store the data in S3
    s3 = boto3.client('s3')
    object_key = s3_prefix + datetime.now().strftime('%Y-%m-%d-%H-%M-%S') + '.tx
    s3.put_object(Bucket=bucket_name, Key=object_key, Body=data)

# Run the job once immediately to test it
scrape_and_store_data()

# Schedule the job to run every day at 2AM UTC+07
while True:
    now = datetime.now(timezone)
    if now >= scheduled_time:
        scrape_and_store_data()
        scheduled_time += timedelta(days=1)
```

Libraries and functions in PyQuery.

PyQuery is a Python library used for web scraping and parsing HTML documents. It is similar to jQuery, a popular JavaScript library for web development. Some of the key features of PyQuery include:

1. jQuery syntax: PyQuery uses the same syntax as jQuery for selecting and manipulating HTML elements. This makes it easy for developers who are familiar with jQuery to use PyQuery.
2. Easy to use: PyQuery provides a simple and intuitive API for web scraping. It is easy to learn and use for developers of all skill levels.
3. Supports parsing HTML and XML: PyQuery can be used to parse both HTML and XML documents.
4. CSS3 selector support: PyQuery supports CSS3 selectors for selecting HTML elements.

Some of the common functions in PyQuery are:

1. `pq()` : This function is used to create a PyQuery object from an HTML document or a string of HTML.
2. `find()` : This function is used to find HTML elements within a PyQuery object.
3. `eq()` : This function is used to select a specific element from a PyQuery object based on its index.
4. `text()` : This function is used to extract the text content of an HTML element.
5. `attr()` : This function is used to get the value of an attribute of an HTML element.
6. `html()` : This function is used to get the HTML content of an element.
7. `remove()` : This function is used to remove an element from the PyQuery object.

8. `addClass()` : This function is used to add a CSS class to an HTML element.
9. `removeClass()` : This function is used to remove a CSS class from an HTML element.
10. `append()` : This function is used to add HTML content to an element.

These are just a few of the functions provided by PyQuery. There are many more functions available that make it easy to scrape and manipulate HTML documents using Python.

Example of Selenium in Python to extract data from a specific website and store it in AWS S3, and then schedule the job to run every day at 2AM UTC+07

Selenium is a popular automation testing framework that can also be used for web scraping. It provides a variety of functions and libraries for interacting with web pages, including finding elements, filling out forms, and clicking buttons.

- [ ] Here's an example of using Selenium in Python to extract data from a specific website and store it in AWS S3, and then scheduling the job to run every day at 2AM UTC+07.

First, we need to install the necessary packages:

```
pip install selenium
pip install boto3
pip install pytz
```

```
from selenium import webdriver
from selenium.webdrive.chrome.options import Options
from datetime import datetime, timedelta
import time
import boto3

# set up AWS S3 connection
s3 = boto3.client('s3')
# set up Selenium webdriver with options
chrome_options = Options()
chrome_options.add_argument('--headless')
chrome_options.add_argument('--no-sandbox')
chrome_options.add_argument('--disable-dev-shm-usuage')
```

```
# driver = webdriver.Chrome
driver = webdriver.Chrome(options= chrome_options)
# navigate to website and log in
driver.get('https://www.example.com/login')
username = driver.find_element_by_id('username')
password = driver.find_element_by_id('password')
username.send_keys('your_username')
password.send_keys('your_password')
login_button = driver.find_element_by_name('login')
login_button.click()
# navigate to page with data to scrape
driver.get('https://www.example.com/data')
# scrape data and store in list
data = []
table = driver.find_element_by_xpath('//table[@class="data-table"]')
rows = table.find_elements_by_tag_name('tr')
for row in rows:
    cells = row.find_elements_by_tag_name('td')
    data.append([cell.text for cell in cells])
# write data to AWS S3
bucket_name = 'your_bucket_name'
key = 'data_{}.csv'.format(datetime.utcnow().strftime('%Y%m%d'))
csv_data = '\n'.join([','.join(row) for row in data])
s3.put_object(Bucket=bucket_name, Key=key, Body=csv_data)
# quit webdriver
driver.quit()
# schedule job to run every day at 2AM UTC+07
while True:
    now = datetime.utcnow() + timedelta(hours=7)
    if now.hour == 2 and now.minute == 0:
# run scraping and upload to S3 code here
        time.sleep(60)
  # wait a minute to avoid running multiple times in one day
    else:
        time.sleep(60)
    # wait a minute and check again
```

Note that this example is a starting point and would need to be modified. Thus, running a job every day at a specific time typically requires running the code on a server or cloud service, so you may want to look into options such as AWS Lambda or Google Cloud Functions to handle the scheduling and executions of the code.

Libraries and functions in PyQuery.

Selenium is a popular Python library used for web automation and web scraping. It provides various methods and functions to interact with web pages and extract data. Here are some of the commonly used functions and methods in Selenium:

1. `webdriver` : This function is used to start a new web driver instance for the specified browser. For example, `webdriver.Firefox()` creates a new Firefox web driver instance.
2. `get()` : This method is used to navigate to a specific URL. For example, `driver.get("https://www.google.com")` navigates to the Google homepage.
3. `find_element_by_*()` : These methods are used to find elements on a web page using different types of locators such as ID, class name, name, CSS selector, and XPath. For example, `driver.find_element_by_id("username")` finds the element with ID "username".
4. `send_keys()` : This method is used to send text to an input field. For example, `element.send_keys("hello")` sends the text "hello" to the input field.
5. `click()` : This method is used to click on an element. For example, `element.click()` clicks on the element.
6. `execute_script()` : This method is used to execute JavaScript code on a web page. For example, `driver.execute_script("alert('hello')")` displays an alert box with the message "hello".
7. `get_attribute()` : This method is used to get the value of an attribute of an element. For example, `element.get_attribute("href")` gets the value of the "href" attribute of the element.
8. `page_source` : This property is used to get the HTML source code of the current page. For example, `html = driver.page_source` gets the HTML source code of the current page.
9. `implicitly_wait()` : This method is used to set a default waiting time for the driver to wait for an element to appear on the page before throwing an exception. For example, `driver.implicitly_wait(10)` sets the waiting time to 10 seconds.

These are just a few of the functions and methods provided by Selenium. There are many more functions and methods available for various web automation and web scraping tasks.