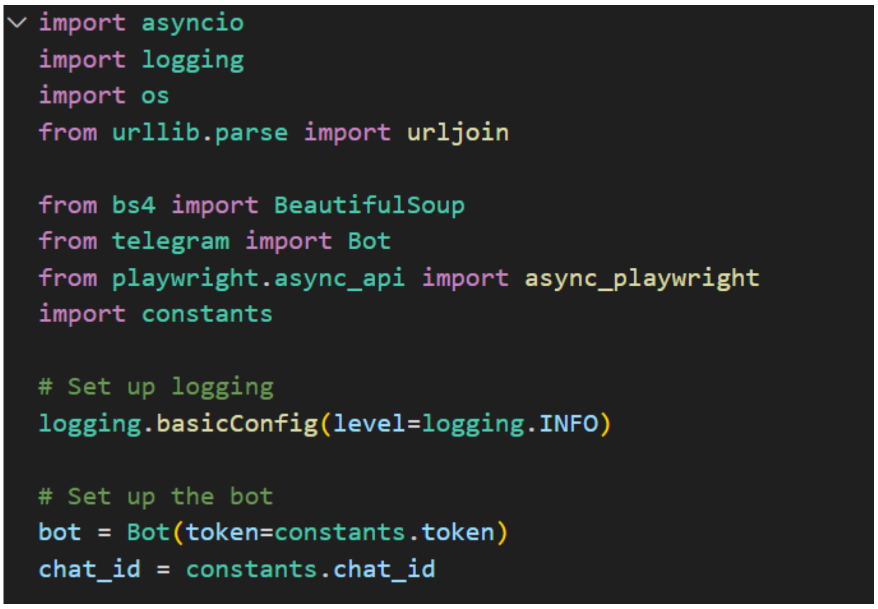
Documentation for telegram bot



```python
import asyncio
import logging
import os
from urllib.parse import urljoin

from bs4 import BeautifulSoup
from telegram import Bot
from playwright.async_api import async_playwright
import constants

# Set up logging
logging.basicConfig(level=logging.INFO)

# Set up the bot
bot = Bot(token=constants.token)
chat_id = constants.chat_id
```

Purpose of each library:

Asyncio: Allows the bot to send telegram messages while still processing the next article without freezing. Not replaceable with the time module because time modules will freeze the whole program (i.e. blocking)

Logging: Used to send error messages to the terminal if anything goes wrong (e.g. Scraper fails to fetch an article

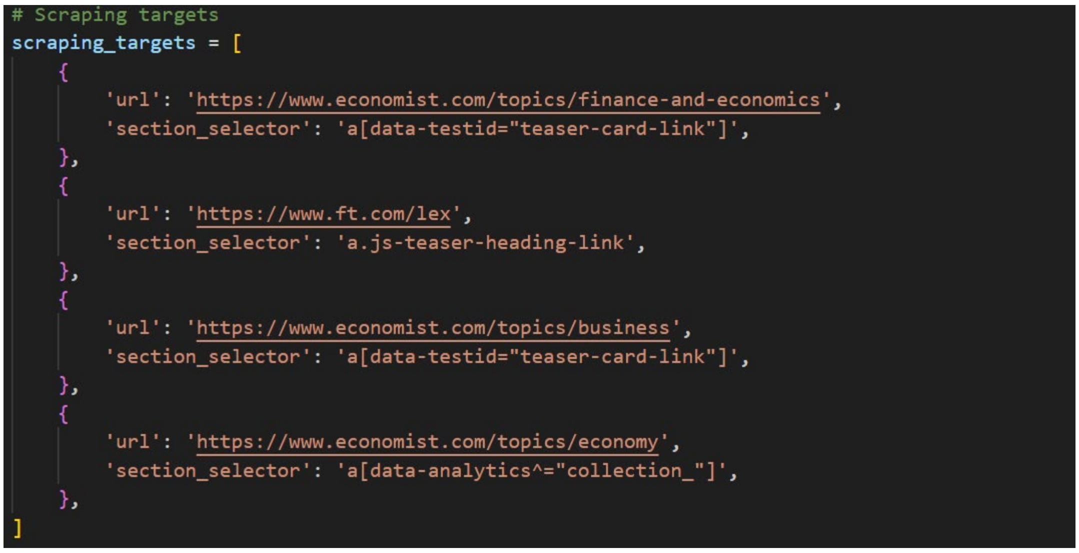
Os: Used to store links in a .txt file to prevent repeated news articles

Urllib.parse: to create clickable links

BeautifulSoup: Used to parse the HTML of the webpage so that the news article links and headline can be extracted

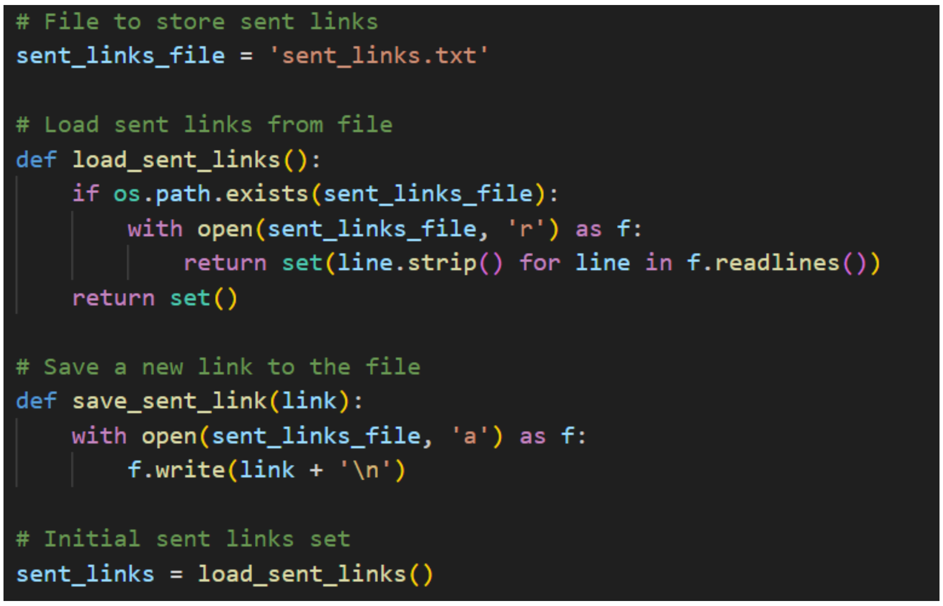
Telegram & Constants: Used to help initialize the telegram bot

Async_playwright: To launch and control a Chromium browser asynchronously, allowing the scraper to fetch rendered HTML without blocking the Python event loop.


```python
# Scraping targets
scraping_targets = [
    {
        'url': 'https://www.economist.com/topics/finance-and-economics',
        'section_selector': 'a[data-testid="teaser-card-link"]',
    },
    {
        'url': 'https://www.ft.com/lex',
        'section_selector': 'a.js-teaser-heading-link',
    },
    {
        'url': 'https://www.economist.com/topics/business',
        'section_selector': 'a[data-testid="teaser-card-link"]',
    },
    {
        'url': 'https://www.economist.com/topics/economy',
        'section_selector': 'a[data-analytics^="collection_"]',
    },
]
```

Scraping targets:

Upon inspection of the webpage's HTML, I found the common CSS selector (used to find both the title and link to the article). The URL provided is the section I am interested in


```python
# File to store sent links
sent_links_file = 'sent_links.txt'

# Load sent links from file
def load_sent_links():
    if os.path.exists(sent_links_file):
        with open(sent_links_file, 'r') as f:
            return set(line.strip() for line in f.readlines())
    return set()

# Save a new link to the file
def save_sent_link(link):
    with open(sent_links_file, 'a') as f:
        f.write(link + '\n')

# Initial sent links set
sent_links = load_sent_links()
```

Preventing duplicate articles

To prevent duplicate articles, I decided to store a .txt in the same folder with the links that have already been sent to the telegram chat. To do this, I needed **save_sent_links()** and **load_sent_links().**

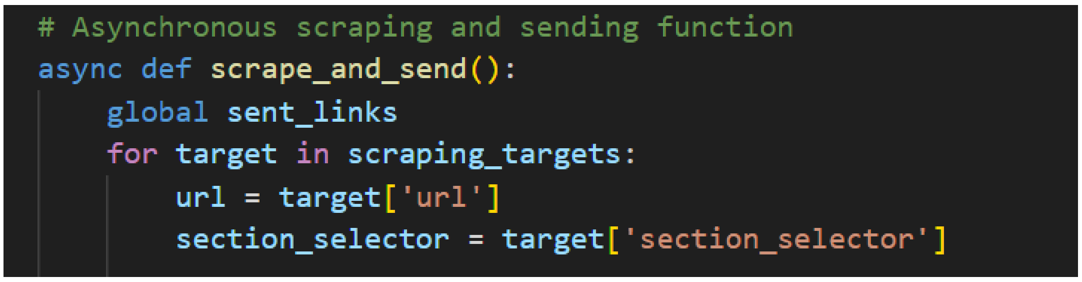
Save_sent_links(): This function takes in an argument (link of the news article) and writes it into the .txt file

Load_sent_links(): The first step this function does is to read the sent_links.txt file to gain a memory of the links previously sent and returns a set of those links

Main system design

The scraper is designed as an asynchronous system combining Playwright and BeautifulSoup to fetch and parse the HTML. The main **scrape_and_send** function iterates through **scraping_targets** asynchronously, awaiting HTML fetch and telegram API calls.

The **sent_links** .txt file is made accessible in this function using global keyword. The function starts a loop and extracts the URL and section selector (used to find both the title and link to the article).

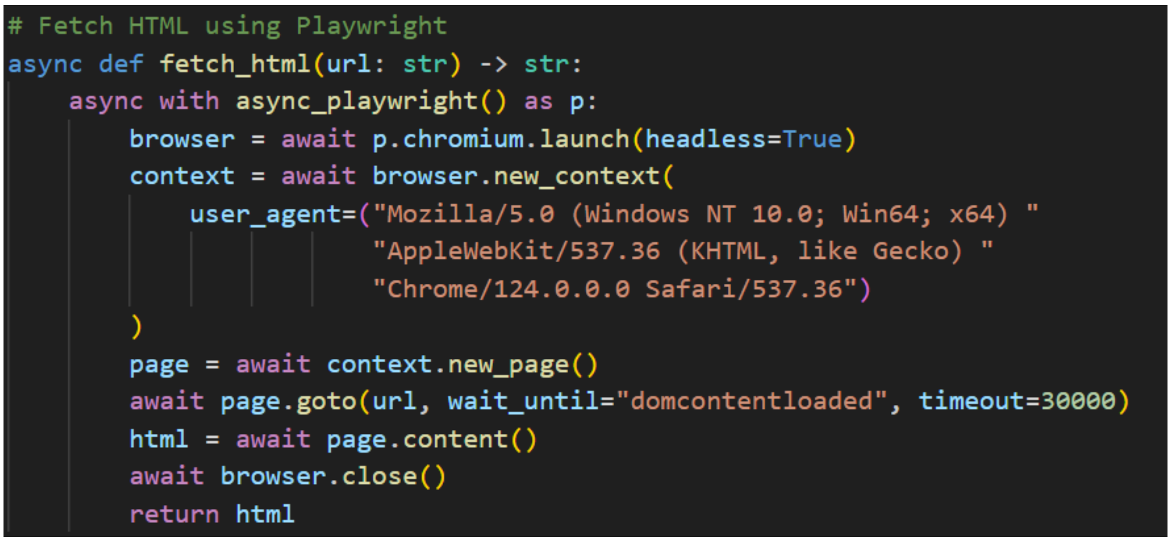


```python
# Asynchronous scraping and sending function
async def scrape_and_send():
    global sent_links
    for target in scraping_targets:
        url = target['url']
        section_selector = target['section_selector']
```

For each scraping target, the code will create a new playwright context manager, within which each will contain a browser, context and pages.

Browser: Launch a Chromium browser
Context: Profile for the browser
Page: Tab in the browser. Page.goto and page.content are functions that work on the page level to load the page (JavaScript) and return the HTML for that page.

Here, the function is defined with type hinting to ensure argument of string type url of the page and return a string type of the page's HTML.
The awaits also play a key function in ensuring everything is executed in the right order.



```python
# Fetch HTML using Playwright
async def fetch_html(url: str) -> str:
    async with async_playwright() as p:
        browser = await p.chromium.launch(headless=True)
        context = await browser.new_context(
            user_agent=("Mozilla/5.0 (Windows NT 10.0; Win64; x64) "
                        "AppleWebKit/537.36 (KHTML, like Gecko) "
                        "Chrome/124.0.0.0 Safari/537.36")
        )
        page = await context.new_page()
        await page.goto(url, wait_until="domcontentloaded", timeout=30000)
        html = await page.content()
        await browser.close()
        return html
```

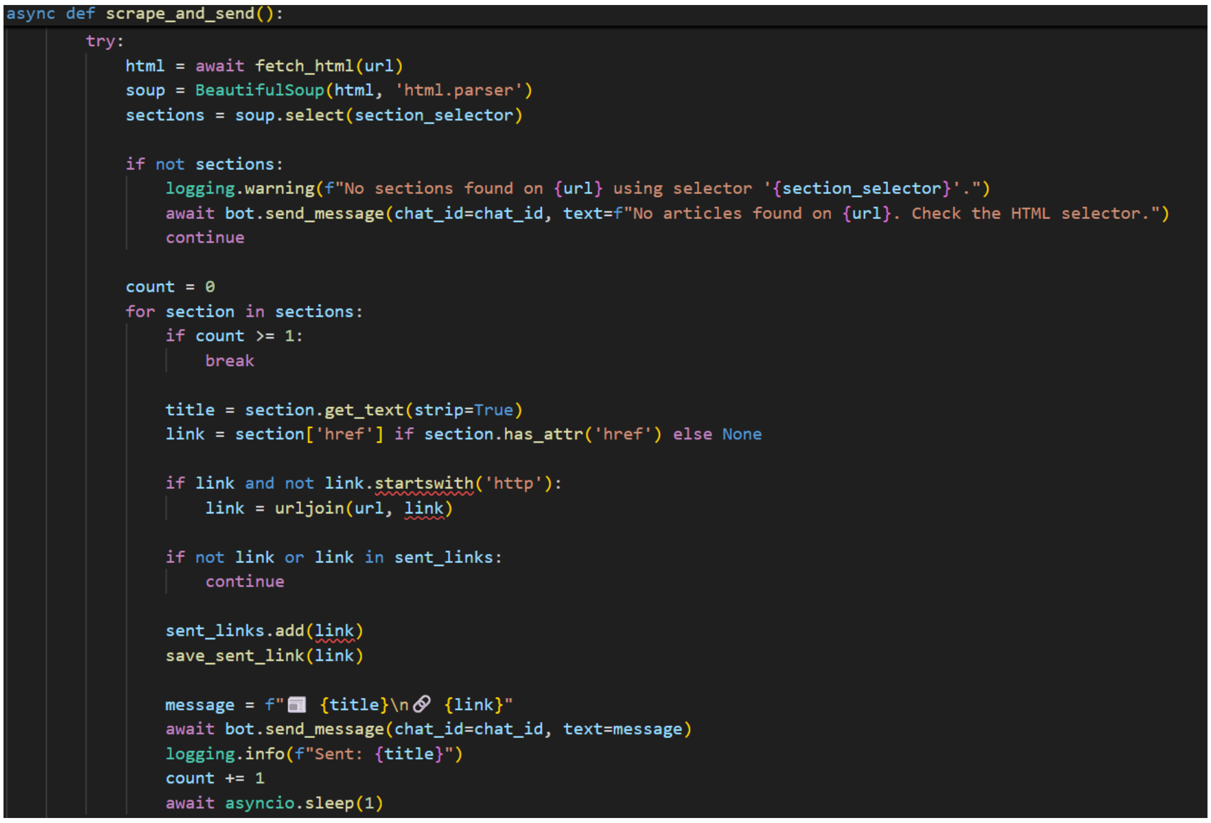
<u>Try-except structure and asynchronous flow:</u>

The entire process of scraping is wrapped in a try-except block. If any target URL fails, the loop goes to the except block, then goes onto the next target.

The scraper makes use of a sequential asynchronous design. Through testing, concurrent async design would make the program faster due to overlaps in page navigation & HTML fetching.

await tells the coroutine to pause at that point until the awaited asynchronous operation completes, allowing the event loop to run other tasks in the meantime. In this case, it could be page navigation, HTML fetching and sending telegram messages. This ensures the coroutine's order of scraping activities is maintained while asynchronously working the CPU on other tasks.

<u>Try:</u>



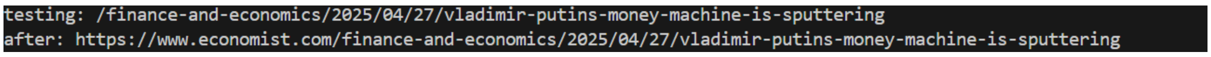
```python
async def scrape_and_send():
        try:
            html = await fetch_html(url)
            soup = BeautifulSoup(html, 'html.parser')
            sections = soup.select(section_selector)

            if not sections:
                logging.warning(f"No sections found on {url} using selector '{section_selector}'.")
                await bot.send_message(chat_id=chat_id, text=f"No articles found on {url}. Check the HTML selector.")
                continue

            count = 0
            for section in sections:
                if count >= 1:
                    break

                title = section.get_text(strip=True)
                link = section['href'] if section.has_attr('href') else None

                if link and not link.startswith('http'):
                    link = urljoin(url, link)

                if not link or link in sent_links:
                    continue

                sent_links.add(link)
                save_sent_link(link)

                message = f"📰 {title}\n🔗 {link}"
                await bot.send_message(chat_id=chat_id, text=message)
                logging.info(f"Sent: {title}")
                count += 1
                await asyncio.sleep(1)
```

The sequence of scraping activities is as such:
1. Playwright is used to fetch the HTML
2. Beautiful soup is then used to parse the HTML content, using the CSS selector to find the desired articles within that HTML. The If not section conditional statements ensure that a warning will be thrown if no articles were found using that selector

A **count** variable is then introduced to limit the amount of articles sent from each website to prevent overwhelming notifications.

The title and article link are then extracted into the **title** and **link** variables using the BeautifulSoup library (.get_text() and 'href' attribute).

The first **if-statement** is to convert the link into a clickable https URL:

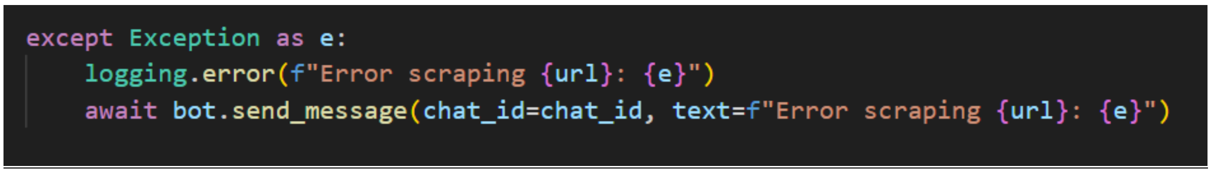


```
testing: /finance-and-economics/2025/04/27/vladimir-putins-money-machine-is-sputtering
after: https://www.economist.com/finance-and-economics/2025/04/27/vladimir-putins-money-machine-is-sputtering
```

The second **if-statement** is to leverage the memory (sent_links) to prevent sending duplicate links. I then saved the extracted link from that scraping target into the memory file by calling **save_sent_links(link).**

An f-string is then use to craft the message with the news link and the title and bot.send_message is used to send the message to telegram. The count variable is incremented by 1 to prevent spamming of articles, a logging.info is triggered to create a log of the article sent using the telegram API and a 1 second asyncio delay is implemented.

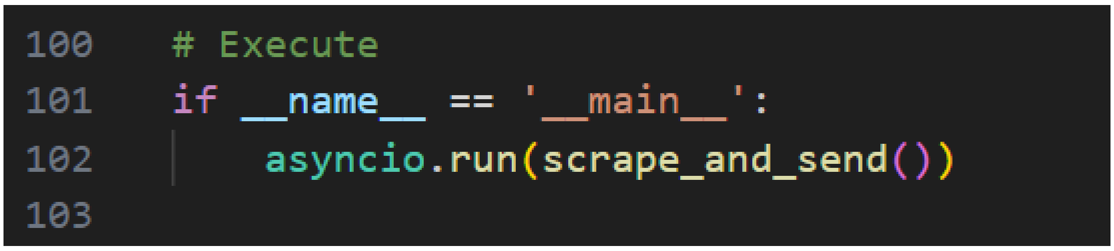
<u>Except:</u>



```python
except Exception as e:
    logging.error(f"Error scraping {url}: {e}")
    await bot.send_message(chat_id=chat_id, text=f"Error scraping {url}: {e}")
```

This part of the code makes use of the logging and telegram libraries to record the error encountered, e.g. sending HTML request, parsing HTML, extracting the sections, getting the title & link and sending the tele message.

<u>Except block:</u>



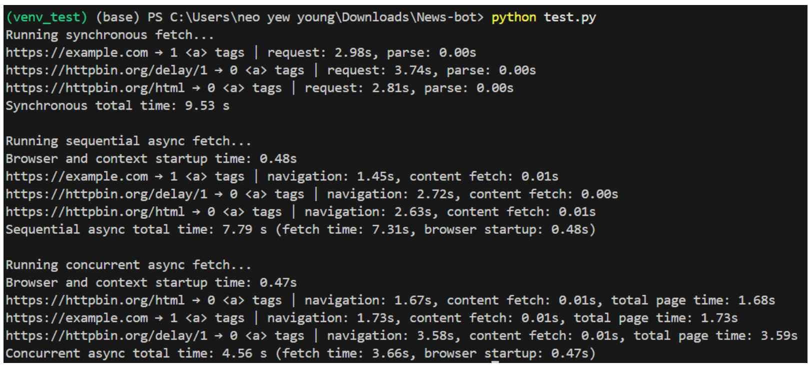
```python
# Execute
if __name__ == '__main__':
    asyncio.run(scrape_and_send())

```

Lastly, the **scrape_and_send** function is executed

Sources:

- ChatGPT
- https://medium.com/@samuelmideksa4/web-scraping-using-python-c2924e3f0924

Appendix:



```
(venv_test) (base) PS C:\Users\neo yew young\Downloads\News-bot> python test.py
Running synchronous fetch...
https://example.com → 1 <a> tags | request: 2.98s, parse: 0.00s
https://httpbin.org/delay/1 → 0 <a> tags | request: 3.74s, parse: 0.00s
https://httpbin.org/html → 0 <a> tags | request: 2.81s, parse: 0.00s
Synchronous total time: 9.53 s

Running sequential async fetch...
Browser and context startup time: 0.48s
https://example.com → 1 <a> tags | navigation: 1.45s, content fetch: 0.01s
https://httpbin.org/delay/1 → 0 <a> tags | navigation: 2.72s, content fetch: 0.00s
https://httpbin.org/html → 0 <a> tags | navigation: 2.63s, content fetch: 0.01s
Sequential async total time: 7.79 s (fetch time: 7.31s, browser startup: 0.48s)

Running concurrent async fetch...
Browser and context startup time: 0.47s
https://httpbin.org/html → 0 <a> tags | navigation: 1.67s, content fetch: 0.01s, total page time: 1.68s
https://example.com → 1 <a> tags | navigation: 1.73s, content fetch: 0.01s, total page time: 1.73s
https://httpbin.org/delay/1 → 0 <a> tags | navigation: 3.58s, content fetch: 0.01s, total page time: 3.59s
Concurrent async total time: 4.56 s (fetch time: 3.66s, browser startup: 0.47s)
```