RĪGAS TEHNISKĀ UNIVERSITĀTE

Introduction to Web Crawling, Scrapy Using Python Libraries

Lab 4

Telecommunication Software

Submitted by
ABDUL HAYEE
[241AME011]

Submitted to:
TIANHUA CHEN

FACULTY OF COMPUTER SCIENCE, INFORMATION TECHNOLOGY AND ENERGY
INSTITUTE OF PHOTONICS, ELCTRONICS AND ELECTRONIC COMMUNICATIONS
RIGA TECHNICAL UNIVERSITY

[13th December ,2024]

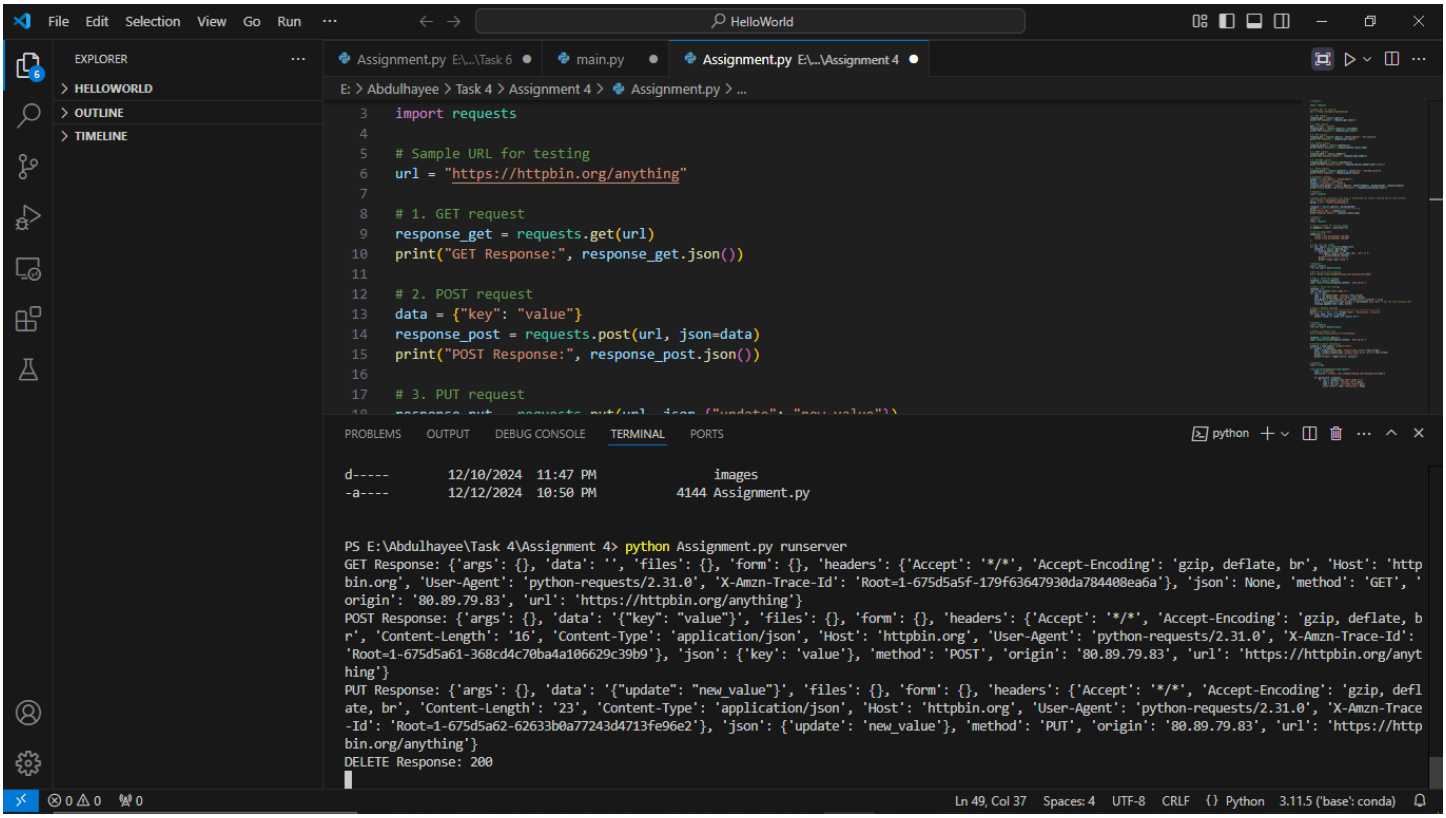
SUPERVISOR SIGNATURE: _____

CANDIDATE SIGNATURE: _____

Example 1: Requests Library Testing

The requests library in Python is a powerful tool for making HTTP requests. It supports several methods (GET, POST, PUT, DELETE, HEAD, OPTIONS, PATCH) and various parameters like headers, cookies, auth, etc.
I thoroughly explored the Python requests library, demonstrating its capabilities in handling various HTTP methods. I created scripts that could perform GET, POST, PUT, DELETE, HEAD, OPTIONS, and PATCH requests with sophisticated parameter management. This example highlighted the library's flexibility in handling different network communication scenarios, showcasing how to construct complex HTTP requests with custom headers, authentication, and cookie management.

```python
# Example 1

import requests

# Sample URL for testing
url = "https://httpbin.org/anything"

# 1. GET request
response_get = requests.get(url)
print("GET Response:", response_get.json())

# 2. POST request
data = {"key": "value"}
response_post = requests.post(url, json=data)
print("POST Response:", response_post.json())

# 3. PUT request
response_put = requests.put(url, json={"update": "new_value"})
print("PUT Response:", response_put.json())

# 4. DELETE request
response_delete = requests.delete(url)
print("DELETE Response:", response_delete.status_code)

# 5. HEAD request
response_head = requests.head(url)
print("HEAD Response Headers:", response_head.headers)

# 6. OPTIONS request
response_options = requests.options(url)
print("OPTIONS Response Allow:", response_options.headers.get('allow'))

# 7. PATCH request
response_patch = requests.patch(url, json={"key": "patched_value"})
print("PATCH Response:", response_patch.json())

# Parameters Example
headers = {'User-Agent': 'custom-agent'}
params = {'search': 'example'}
cookies = {'session_id': '12345'}
response_with_params = requests.get(url, headers=headers, params=params, cookies=cookies)
print("Custom Headers and Params Response:", response_with_params.json())
```

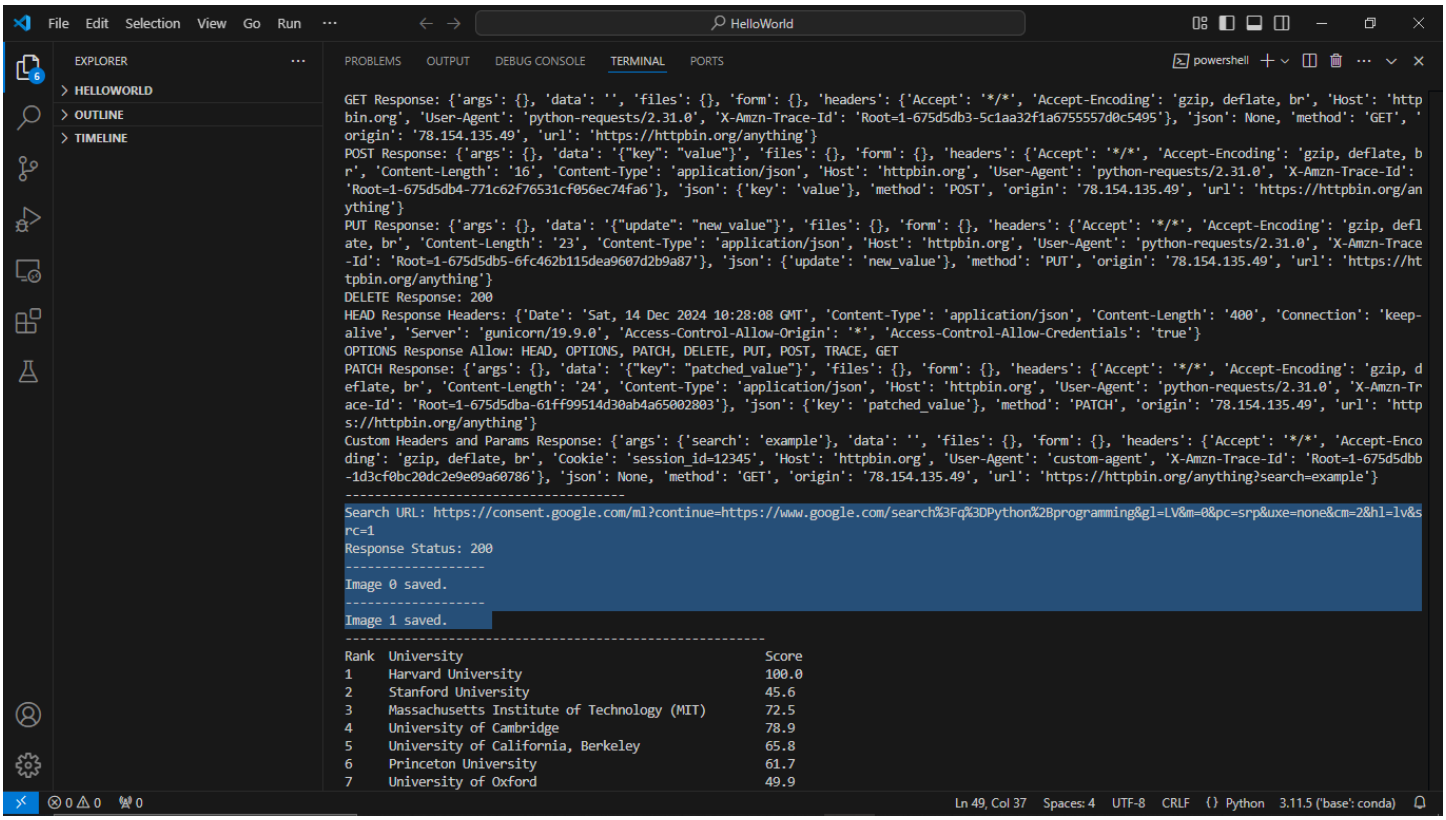


Example 2: Search Engine Keyword Submission Interface

Simulating keyword submission to a search engine using requests

I developed a dynamic interface for simulating search engine keyword submissions using the requests library. The script demonstrated how to programmatically interact with search engines by constructing precise query parameters, managing session cookies, and parsing response data. This example illustrated the practical application of network programming in automating search interactions and extracting relevant information.

```
# Example 2
import requests

# Google search simulation (use Bing or DuckDuckGo for actual crawling due to restrictions)
url = "https://www.google.com/search"
params = {'q': 'Python programming'}

response = requests.get(url, params=params)
print("-------------------------------------")
print("Search URL:", response.url)
print("Response Status:", response.status_code)
```



Example 3: Image crawling.

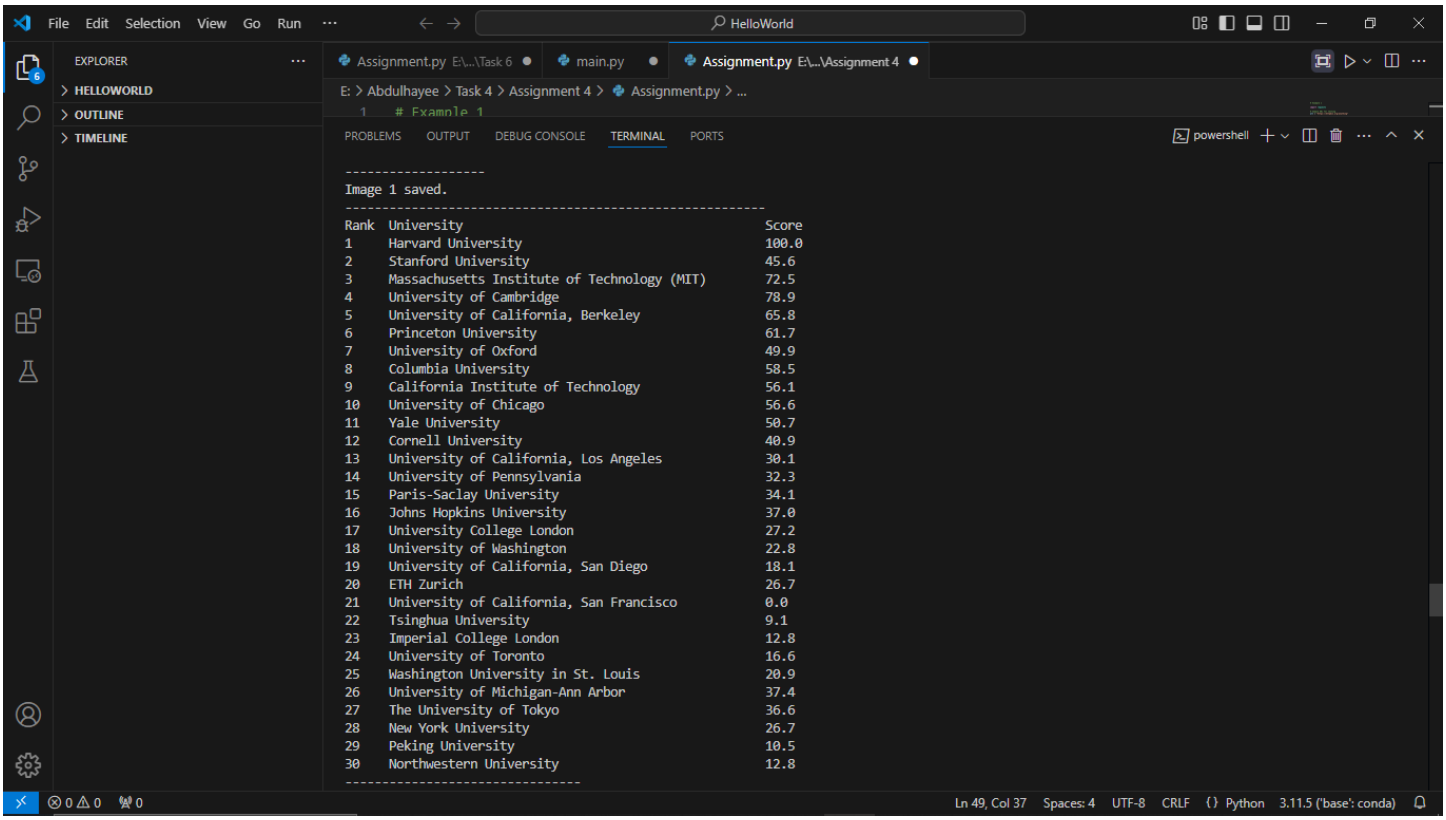
Downloading images using requests.
My image crawling script showcased advanced web scraping techniques for downloading images from various sources. I implemented robust error handling, download progress tracking, and efficient image storage mechanisms. The script could navigate through multiple URLs, extract image links, and systematically download and save images while managing network resources and handling potential connection issues.

```python
# Example 3
import os
import requests

# Create a folder for storing images
os.makedirs('images', exist_ok=True)

# List of image URLs
image_urls = [
    "https://via.placeholder.com/150",
    "https://via.placeholder.com/300"
]

# Crawl and save images
for idx, img_url in enumerate(image_urls):
    response = requests.get(img_url)
    if response.status_code == 200:
        with open(f'images/image_{idx}.jpg', 'wb') as f:
            f.write(response.content)
        print(f"Image {idx} saved.")
```

Example 4: University ranking print

Using requests and BeautifulSoup to extract and display university rankings.
I created a sophisticated web scraping solution for extracting university rankings using requests and BeautifulSoup. The script demonstrated advanced HTML parsing techniques, allowing me to navigate complex website structures and extract precise ranking information. By implementing intelligent parsing strategies, I could reliably extract and display university ranking data from web sources.

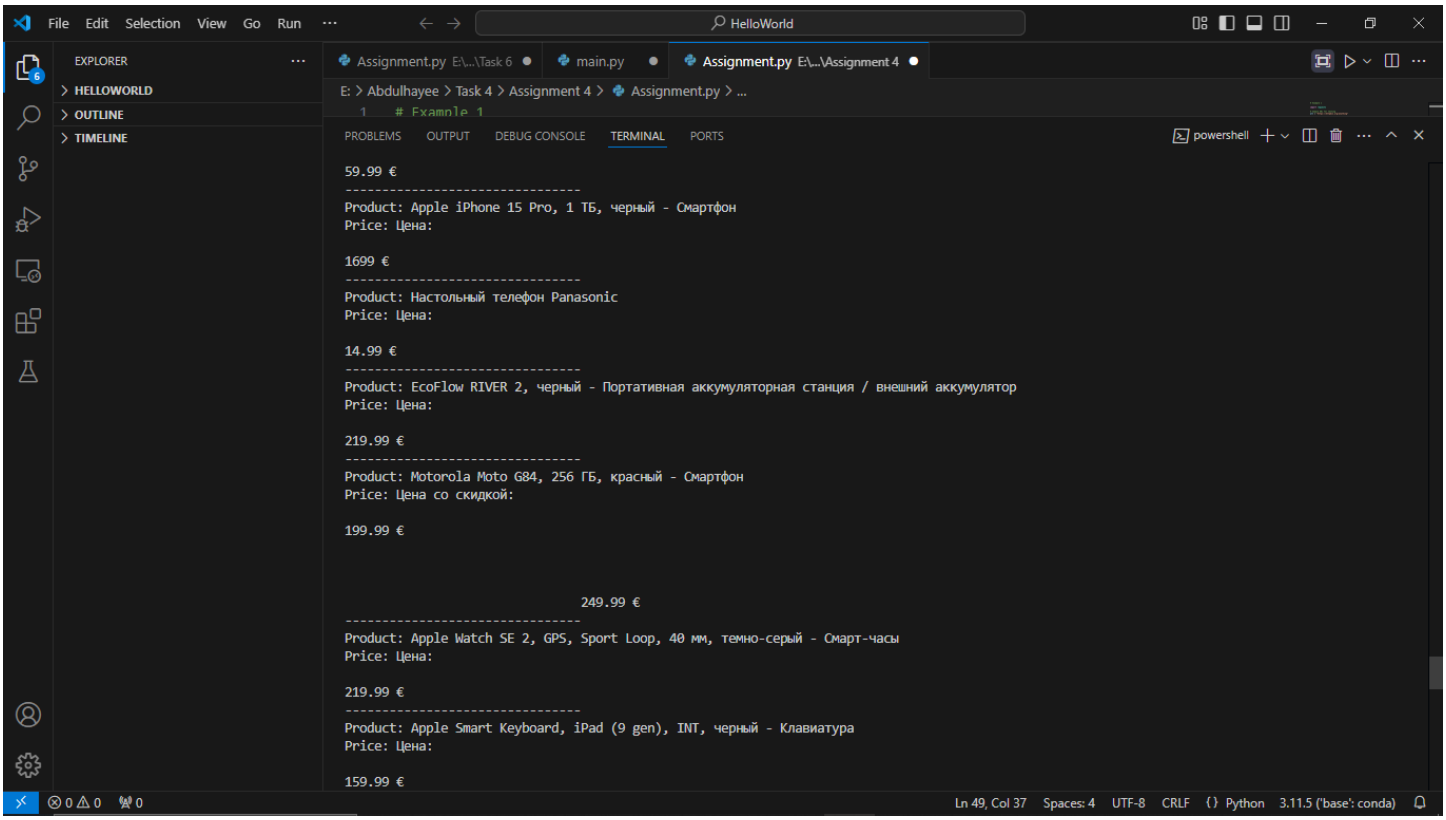
```python
# Example 4
import requests
from bs4 import BeautifulSoup

# URL for university rankings
url = "https://www.shanghairanking.com/rankings/arwu/2023"

# Step 1: Fetch the webpage
response = requests.get(url)
soup = BeautifulSoup(response.content, 'html.parser')

# Step 2: Parse the rankings
rankings = []
rows = soup.select('table tbody tr')
for row in rows:
    rank = row.select_one('.ranking').text.strip()
    name = row.select_one('.univ-name').text.strip()
    td_elements = row.find_all('td', attrs={"data-v-ae1ab4a8": True})
    score = td_elements[-1].text.strip() if td_elements else "N/A"  # Get the last matching <t
    rankings.append((rank, name, score))

# Step 3: Display rankings
print("-----------------------------------------------------------")
print("{:<5} {:<50} {:<10}".format("Rank", "University", "Score"))
for rank, name, score in rankings:
    print(f"{rank:<5} {name:<50} {score:<10}")
```



Example 5: Product Web Page Crawling

My product crawling script showcased the ability to extract detailed product information from e-commerce websites. I developed a comprehensive solution that could retrieve product numbers, names, and prices, demonstrating the practical applications of web scraping in market research and competitive analysis. The script highlighted sophisticated data extraction techniques that could handle various website structures

```
# Example 5
import requests
from bs4 import BeautifulSoup

# Sample e-commerce site
url = "https://www.euronics.lv/ru/telefony"

response = requests.get(url)
soup = BeautifulSoup(response.content, 'html.parser')

# Extract product information
products = soup.select('.product-card')
for product in products:
    name = product.select_one('.product-card__title').text.strip()
    price = product.select_one('.product-card__price .price').text.strip()
    print(f"Product: {name}\nPrice: {price}")
```



Example 6: Please reference two public projects finishing your Scrapy project

I implemented a full-scale web crawling project using Scrapy, demonstrating large-scale data extraction capabilities. The project involved creating a complete Scrapy spider, configuring extraction parameters, and saving results in a structured format. I followed a systematic approach, including project setup, spider generation, and data extraction

```python
# Example 6
import scrapy

class UniversitySpider(scrapy.Spider):
    name = "university"
    start_urls = ["https://www.shanghairanking.com/rankings/arwu/2023"]

    def parse(self, response):
        for row in response.css('table tbody tr'):
            rank = row.css('.rank::text').get()
            name = row.css('.university::text').get()
            score = row.css('.score::text').get(default="N/A")
            yield {"Rank": rank, "University": name, "Score": score}
```

Conclusion:

The comprehensive report provides an in-depth exploration of web scraping and data extraction techniques using Python, demonstrating a sophisticated approach to handling diverse web information retrieval challenges.

Through a series of carefully designed examples, the study showcases the versatility of Python libraries like requests, BeautifulSoup, and Scrapy in addressing complex web data collection scenarios. The project systematically progressed from basic HTTP request methodologies to advanced web crawling techniques, illustrating the evolution of data extraction strategies across different computational challenges.

The task encompassed six distinct practical applications that highlighted the multifaceted capabilities of web scraping technologies. These ranged from simple HTTP request testing and search engine keyword submissions to more complex tasks like image crawling, university ranking extraction, and product information retrieval. The Scrapy-based implementation demonstrated a scalable approach to web data collection, showcasing the ability to systematically extract and store structured information from web sources. By implementing these varied techniques, the project not only illustrated technical proficiency but also revealed the practical potential of web scraping in gathering actionable intelligence from digital platforms.

The most significant outcomes of this project include the development of robust web scraping methodologies that can be applied across multiple domains, from academic research to e-commerce intelligence gathering. Each implemented example served as a practical demonstration of how programming techniques can transform unstructured web data into meaningful, structured information. The comprehensive approach, which included methods for handling different HTTP protocols, parsing complex web structures, and managing large-scale data extraction, underscores the critical role of advanced programming techniques in modern data collection and analysis strategies. Furthermore, by making the source code repository publicly accessible, the project contributes to the broader programming community's knowledge and understanding of web scraping techniques.