MSc. Data Science

Written Assignment: DLMDSPWP01 – Programming with Python

Topic: Extracting Insights from the Web - Web Scraping with Python

December 2023

Noel Tete Tetteh

Matriculation Number: 4232194

Tutor: Amna Khan

Table of Contents

Introduction

In today's data-driven world, the ability to extract meaningful insights from vast amounts of information has become a crucial skill for analysts. Web scraping, the process of extracting data from websites, has emerged as a powerful tool for gathering and analyzing this information. This write-up delves into the world of web scraping using Python, providing a step-by-step approach for beginners to grasp this valuable technique.

Rationale for Topic Selection

Web scraping has become increasingly relevant in recent years due to the exponential growth of data available online. According to a 2023 report by Statista, the web scraping market is expected to reach $2.4 billion by 2028 (Statista, 2023). Some key findings in the report are:

- The global web scraping market is projected to grow at a Compound Annual Growth Rate (CAGR) of 18.2% from 2022 to 2028.
- The market is being driven by the increasing demand for data-driven insights, the growth of e-commerce, and the rise of API-based web scraping solutions.

Also, in another blog by Popupsmart entitled “15 Best Web Scraping Tools in 2024 to Extract Online Data” below are some key findings (Ozsahan, 2023):

- Bright Data Web Scraper IDE is the best web scraping tool, followed by Oxylabs Scraper API, Webscraper.io, and Apify.
- Free tools like Parsehub and Scrapy are gaining popularity due to their ease of use and user-friendly interfaces.
- The choice of web scraping tool depends on the specific needs of the project, such as the complexity of the data to be extracted, the frequency of scraping, and the resources available.

Aside from these, according to an article titled “7 Reasons Why Web Scraping is Popular Currently” by HIR Infotech, the following reasons were also listed (Infotech, 2022):

1. Web Scraping Service Provider Progression – The author mentions how in recent times, web scraping service providers have advanced significantly.

2. Quick data extraction – The author mentions how manually scraping web pages does not make much sense in this age, and gives an example of how a web scraper can scrape as much data in a day, as a man could in a whole year.
3. Data reliability – The author talks about the greatest level of accuracy when using reliable web scraping services.
4. Tracking price wars – The writer refers to real-time monitoring of prices by "rival" companies via their websites. A web scraping pricing model would be crucial for this to happen.
5. Tracking the stock market. The use of a web scraper to align completely with the stock market is another big advantage. The writer refers to how such a model web scraper, when automated can add real value to a business.
6. Real estate listing – Imagine having to manually compile information on listed properties in a city. It is not impossible, however, it would simply take too much time. The writer talks here about how businesses use web scraping tools to boost the number of listings of their real estate.

In another article by Guillaume Odier of Captain Data, he also lists several reasons web scraping is important. Key amongst his reasons are innovation; but not just innovation, innovation at the "speed of light" (Odier, 2018). Take Kayak for example, or Skyscanner, DuckDuckGo, Dogpile, and the like, they use data from a web search engine to produce their results; and they do this at the click of a button. This would not be possible without web scraping.

These companies mentioned above, by enabling easier access to web data for everyone, force other companies to enhance their value proposition, and at the end of the day, it is users like you and me who will benefit. Guillaume also talks about how web scraping helps with faster innovation; due to the ability to test and execute new ideas faster (Odier, 2018).

As another example, say you want to build a list of student apartments in Berlin and their prices, but you need a database, this is the perfect opportunity to scrape.

Last but not least, the writer here also talks about marketing automation. Web scraping is a goldmine for marketers (Odier, 2018). On social media such as Facebook, Instagram, Twitter, etc, one can scrape these pages, start extracting followers, follow them, send them direct messages, and advertise their products to a wider audience. Web scraping can also be used to search for companies or individuals on Linkedin, and on Google Maps to find local businesses.

Simply put, the opportunities for web scraping are endless, and that is why this topic is so important in this age of data.

Aim of the Assignment

This assignment aims to provide an introduction to web scraping using Python. The assignment will cover; the fundamentals of web scraping, essential Python libraries for web scraping, practical web scraping techniques, a brief look at advanced web scraping techniques, and real-world applications of web scraping with Python. We will end by linking this write-up to the practical Python assignment.

Topic Boundaries and Necessary Definitions

The scope of this assignment is limited to web scraping using Python. It will focus on extracting data from websites using Python libraries and tools. This paper will not delve into web development, web server architecture, or data analysis techniques beyond the basics.

Key Definitions:

- Web scraping: The process of extracting data from websites (Amos, 2023).
- Web data: Data that is stored and accessible on websites (Amos, 2023).
- Web scraper: A tool or script used to extract data from websites (Amos, 2023).
- HTML: HyperText Markup Language, used to structure web pages (Amos, 2023).
- Parsing: The process of analyzing and interpreting HTML code (Amos, 2023).
- Data extraction: The process of identifying and extracting specific data (Amos, 2023).

Outline of the Structure and Arguments within the Assignment

This assignment will follow a structured approach, starting with an introduction to web scraping and its significance. We will then introduce the essential Python libraries for web scraping, followed by practical examples of web scraping techniques. Advanced web scraping techniques and real-world applications will also be briefly looked at. We will conclude with a few takeaways and link this write-up to the practical Python assignment.

Web Scraping Fundamentals

To begin, let us introduce the fundamentals of web scraping, including its definition, purpose, and ethical considerations. We will discuss the different web scraping techniques, such as manual, semi-automated, and automated methods. The concept of web scraping with Python will be explained, emphasizing its advantages and applications.

So what exactly is web scraping?

There are several definitions out there, however, this one sums it up nicely; Web scraping is the process of extracting data from websites (Amos, 2023). This can be done manually or using automated tools. Web scraping is often used to collect data for research, marketing, or other purposes. That notwithstanding, web scraping also raises several questions that warrant further exploration.

Firstly, web scraping often raises ethical concerns, as it may involve extracting data without the permission of the website owner. When scraping data, extra care must be taken to understand and respect the terms of service of websites being scraped, ensuring that the scraping activities are not intrusive or malicious. Legal implications may also arise due to some jurisdictions having specific laws governing data extraction from websites (Dahito, 2023).

Another question has to do with web scraping’s impact on a website’s performance. For example, excessive or aggressive web scraping can negatively impact a website's performance by overloading its servers and causing it to slow down. When scraping, one must always consider the website’s capacity and use techniques that will not overwhelm their systems (Dahito, 2023).

One final question to consider is how to handle duplicates or missing data. Some websites update periodically, others have typos, inconsistencies in data formats, and the like. Web scrapers must always remember to implement data cleansing techniques to identify and address some of these issues, thereby ensuring that overall data quality is maintained (Dahito, 2023).

Back to learning more about scraping, there are several different web scraping techniques one can employ; these include:

Manual scraping: This involves manually copying and pasting data from websites. For example, I want information on cheap flights from Berlin to Amsterdam, if I navigate to an airline's website, and copy and paste the information I want, I have just scraped data from the web (Webscrape AI, 2023).

Semi-automated scraping: This involves using tools to automate some of the tasks involved in scraping data, such as copying and pasting. Some tools here are Octoparse, Import.io, and ParseHub (Webscrape AI, 2023).

Automated scraping: This involves using tools to automate all of the tasks involved in scraping data. Some of the most common tools here are Scrapy, Beautiful Soup, and Selenium. These tools are embedded in Python and require code to run them (Webscrape AI, 2023).

With the ever-increasing volume of data available online, web scraping has emerged as a powerful tool for gathering and analyzing all sorts of information.

Essential Python Libraries for Web Scraping

The section will introduce the essential Python libraries for web scraping, highlighting their functionalities and applications. The Requests library for making HTTP requests and retrieving web page content, Beautiful Soup for parsing and extracting data from HTML documents, and Pandas for data manipulation and analysis will also be covered.

1. Requests:

The Requests library serves as the foundation for web scraping tasks in Python. It handles HTTP requests, enabling you to retrieve web page content from various sources (Read the Docs, 2023). Imagine you are visiting a website to gather information; Requests acts as your messenger, sending requests to the website and bringing back the desired content.

2. Beautiful Soup:

Once you have the web page content, you will need to parse it, which is where Beautiful Soup comes into play. This library excels at extracting specific data from HTML documents, the building blocks of web pages. Beautiful Soup acts as a decoder, interpreting the HTML code and identifying the data you need (Richardson, 2023).

3. Pandas:

After extracting the data, there is often the need to manipulate and analyze it to extract meaningful insights. This is where Pandas comes in handy. It is the go-to library for data wrangling and provides powerful tools for cleaning, organizing, and transforming data into a format suitable for further analysis (Pandas Documentation — Pandas 2.1.4 Documentation, 2023)

Practical Web Scraping with Python

Now that we are familiar with the essential libraries, let's put them into action. Imagine we want to get information from the internet on the population per country. We can use the following steps below (this list is not definitive, however, it serves the purpose of this assignment):

1. Identify data sources
2. Choose the web scraping method
3. Send requests to the websites
4. Parse the HTML content
5. Identify the specific data elements needed
6. Proceed to extract the data
7. Take care of pagination (if necessary)
8. Clean and format the data
9. Aggregate the data (if necessary)
10. Save the data for further analysis

For the first step, data sources, we have several popular options ie Worldbank, Statista, UNDP, Wikipedia, etc. For this assignment, we would scrape data from Worldometer (Wikipedia defines Worldometer, formerly Worldometers, as a reference website that provides counters and real-time statistics for diverse topics).

Here is a simplified workflow:

```python
# Import necessary libraries
import pandas as pd
import requests
from bs4 import BeautifulSoup

# Define the URL of the website to be scraped
url = 'https://www.worldometers.info/world-population/population-by-country/'

# Send an HTTP GET request to the specified URL
response = requests.get(url)

# Check if the request was successful
if response.status_code == 200:
    # Extract the HTML content from the response
    html_content = response.content

    # Parse the HTML content using BeautifulSoup
    soup = BeautifulSoup(html_content, 'html.parser')

    # Locate the table containing the population data
    table = soup.find('table', {'id': 'example2'})

    # Check if the table exists
    if table is not None:
        # Extract the table header columns
        columns = [header.text.strip() for header in table.find_all('th')]

        # Initialize an empty list to store data rows
        data_rows = []

        # Extract data rows from the table body
        for row in table.find_all('tr')[1:]:
            # Extract data from each table cell
            row_data = [td.text.strip() for td in row.find_all('td')]

            # Append the extracted data row to the list
            data_rows.append(row_data)

        # Create a Pandas DataFrame using the extracted data and columns
        df = pd.DataFrame(data_rows, columns=columns)

        # Print the top 5 rows of the DataFrame
        print(df.head())
    else:
        print("Table not found on the page.")
else:
    print("Error: Unable to retrieve data from the website.")
```

Now let us break it down step-by-step:

- For the first step, we would import the required libraries:

```
# Import the required libraries
import pandas as pd
import requests
from bs4 import BeautifulSoup
```

import pandas as pd: Imports the Pandas library for data analysis and manipulation.

import requests: Imports the Requests library for making HTTP requests to web servers.

from bs4 import BeautifulSoup: Imports the BeautifulSoup library for parsing HTML content.

- Next, we define our URL, of the website we want to scrape:

```
# Define the URL of the website to be scraped
url = 'https://www.worldometers.info/world-population/population-by-country/'
```

- The next step is to send an HTTP GET request to the specified URL

```
# Send an HTTP GET request to the specified URL
response = requests.get(url)
```

This line of code sends an HTTP GET request to the specified URL using the Requests library. HTTP GET requests are used to retrieve data from a specified resource. In this case, the resource is the web page at the url we have defined.

- We now check the status of the request.

```
# Check if the request was successful
if response.status_code == 200:
```

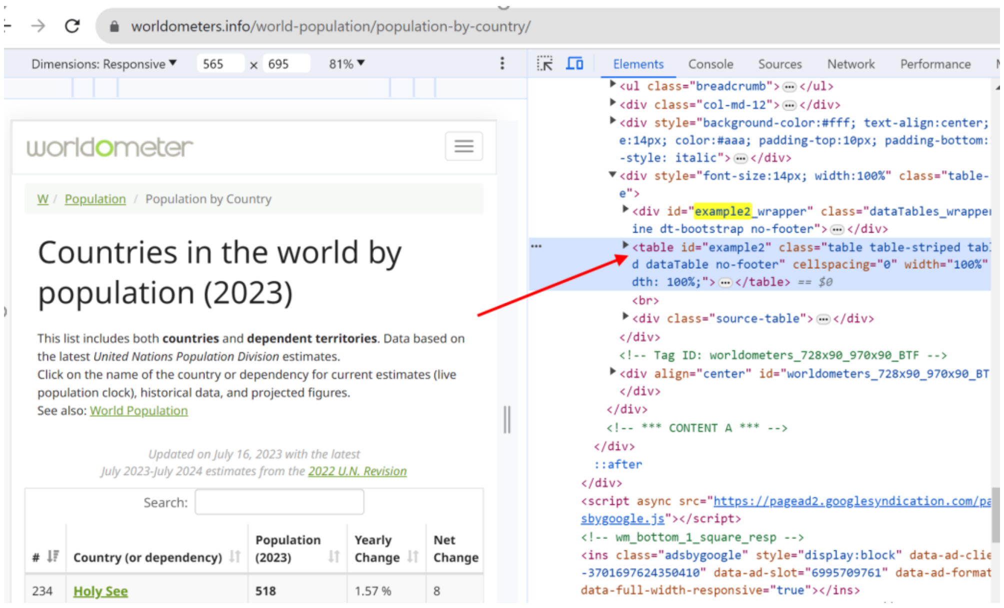
What this line of code does is to check the status code of the HTTP response. A status code of 200 indicates that the request was successful and the resource was found. Other status codes, such as 404 (Not Found) or 500 (Internal Server Error), indicate that there was an error with the request. If the status code is successful, the code does the following under the indented 'If' statement:

```
# Extract the HTML content from the response
html_content = response.content

# Parse the HTML content using BeautifulSoup
soup = BeautifulSoup(html_content, 'html.parser')

# Locate the table containing the population data
table = soup.find('table', {'id': 'example2'})
```

- These 3 lines of code extract, parse and locate.
- The first line of code extracts the HTML content from the HTTP response. (HTML content is the raw HTML code that makes up a web page. It contains the text, images, and other elements that are displayed on the page).
- The second line of code in this block parses the HTML content using Beautiful Soup. (As aforementioned, Beautiful Soup is a library that makes it easy to extract and manipulate data from HTML content) It converts the raw HTML code into a tree-like structure that is easy to navigate, in other words, this line of code prepares the HTML content for further processing.
- For the third line of code, we need to do a couple of things beforehand, and it does get a little bit confusing for first timers, however with practice, it gets easier. (There are different ways this can be done, however for this assignment, this is the way I have decided to do it).
    - First, we need to inspect the url we are working with and get the 'id' of the specific table or data we want to scrape.


worldometers.info/world-population/population-by-country/
Dimensions: Responsive ▾ 565 × 695 81% ▾
Elements
Console
Sources
Network
Performance
worldometer
W / Population / Population by Country
Search:

Countries in the world by population (2023)

This list includes both **countries** and **dependent territories**. Data based on the latest *United Nations Population Division* estimates.
Click on the name of the country or dependency for current estimates (live population clock), historical data, and projected figures.
See also: World Population

Updated on July 16, 2023 with the latest July 2023-July 2024 estimates from the 2022 U.N. Revision

#	Country (or dependency)	Population (2023)	Yearly Change	Net Change
234	Holy See	518	1.57 %	8

```
▸<ul class="breadcrumb">…</ul>
▸<div class="col-md-12">…</div>
▸<div style="background-color:#fff; text-align:center;
 e:14px; color:#aaa; padding-top:10px; padding-bottom:
 -style: italic">…</div>
▾<div style="font-size:14px; width:100%" class="table-
 e">
  ▸<div id="example2_wrapper" class="dataTables_wrappe
   ine dt-bootstrap no-footer">…</div>
  ▸<table id="example2" class="table table-striped tab
   d dataTable no-footer" cellspacing="0" width="100%"
   dth: 100%;">…</table> == $0
   <br>
  ▸<div class="source-table">…</div>
 </div>
 <!-- Tag ID: worldometers_728x90_970x90_BTF -->
▸<div align="center" id="worldometers_728x90_970x90_BT
 </div>
</div>
<!-- *** CONTENT A *** -->
</div>
::after
</div>
<script async src="https://pagead2.googlesyndication.com/pa
sbygoogle.js"></script>
<!-- wm_bottom_1_square_resp -->
<ins class="adsbygoogle" style="display:block" data-ad-clie
-3701697624350410" data-ad-slot="6995709761" data-ad-format
data-full-width-responsive="true"></ins>
```

    - Looking at the image above, our focus is where the red arrow points to; we are looking for the table id. The name of this id is example2, and that is what we use in our code. Once you can locate the id of the data you want to scrape, the rest becomes easier.
- So for the third line of code, the code locates the table with the id 'example2' using the find() method.

The next two lines of code below do the following:

I. Firstly, the code seeks to extract the table header columns from the HTML content using a list comprehension. It does this by retrieving all the <th> elements (these are header cells) from the table object.
II. The 'for loop' then iterates through each <th> element in the list of header cells, and for each header cell, the text.strip method extracts the text content, and removes leading or trailing whitespaces
III. The result is placed in the 'columns' list. This list contains the text content of each header cell, providing the labels for the columns of the table. This information is essential for understanding the meaning of the data extracted from the table rows.

- The next line of code creates or initializes an empty list named 'data_rows'. This list will be used to store the extracted data rows from the table.

```
# Check if the table exists
if table is not None:
   # Extract the table header columns
   columns = [header.text.strip() for header in table.find_all('th')]

   # Initialize an empty list to store data rows
   data_rows = []
```

Now that we have our list to store the data rows, and we have all our column headers, we simply need to go through the table row by row, extract the data and append it to the list. That is what the next lines of code do.

I. The expression 'table.find_all('tr')[1:]:' is used to iterate through each row of a table on a web page. Just as 'th' was used to find the table headers, 'tr' will be used to get the rows. The find_all('tr') method therefore returns a list of all the <tr> elements, which represent rows in an HTML table. The [1:] slice at the end of the expression skips the first row of the table, which is usually the header row.

II. Next, we want the specific data elements, these are represented with ‘td’; hence inside the loop, the row_data = [td.text.strip() for td in row.find_all('td')] expression extracts the data from each cell in the current row, and as before, the strip method removes the trailing whitespaces.

III. The resulting list of cell values is stored in the row_data variable.

IV. Finally, the data_rows.append(row_data) statement appends the extracted data row to the data_rows list. This list accumulates the data from all the rows in the table, creating a collection of data rows that can be further processed or analyzed.

```
# Extract data rows from the table body
for row in table.find_all('tr')[1:]:
    # Extract data from each table cell
    row_data = [td.text.strip() for td in row.find_all('td')]

    # Append the extracted data row to the list
    data_rows.append(row_data)

# Create a Pandas DataFrame using the extracted data and columns
df = pd.DataFrame(data_rows, columns=columns)
```

The next lines of code end the workflow:

I. We finish up with the creation of a Pandas DataFrame from the extracted data rows (data_rows) and columns (columns). In this context, the ‘data_rows’ list contains the extracted data from each table row, and the columns list contains the header labels for each column. By passing these two lists to the pd.DataFrame() constructor, the code creates a DataFrame where each row represents a data record and each column represents a data attribute.

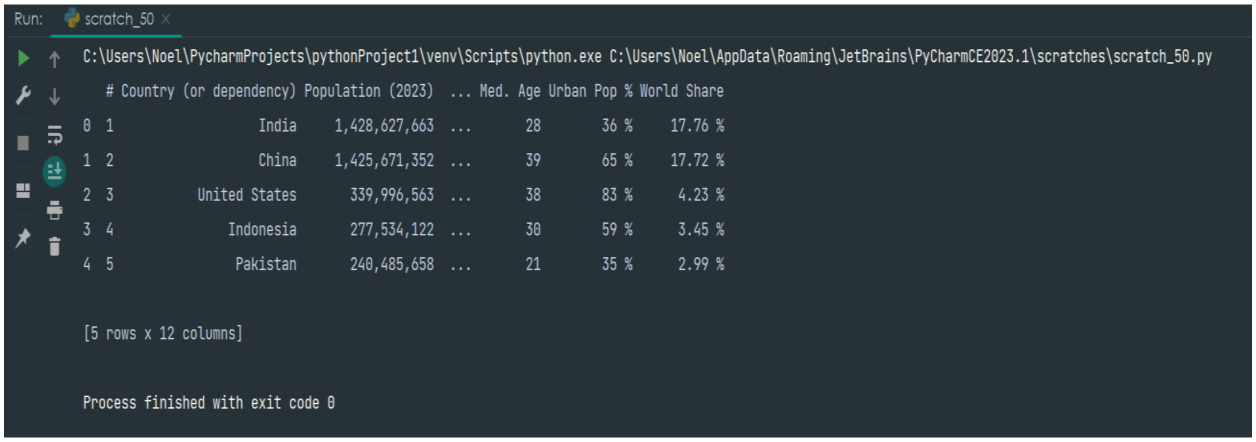
II. The line print(df.head()) prints the top 5 rows of the DataFrame. The head() method displays the first few rows of a DataFrame, which is useful for quickly previewing the data and checking if the extraction process was successful. In this case, it was!

III. The else block handles the situation where the table was not found on the web page. If table is ‘None’, it means the table was not found, and an error message is printed to the console. This error handling ensures that the code can handle scenarios where the target table (the specific table we want) is absent from the web page.

```
    # Create a Pandas DataFrame using the extracted data and columns
        df = pd.DataFrame(data_rows, columns=columns)

        # Print the top 5 rows of the DataFrame
        print(df.head())
    else:
        print("Table not found on the page.")
else:
    print("Error: Unable to retrieve data from the website.")
```

The final results of our web scraper are shown below:



```
C:\Users\Noel\PycharmProjects\pythonProject1\venv\Scripts\python.exe C:\Users\Noel\AppData\Roaming\JetBrains\PyCharmCE2023.1\scratches\scratch_50.py
   # Country (or dependency) Population (2023)  ... Med. Age Urban Pop % World Share
0  1                   India     1,428,627,663  ...       28        36 %     17.76 %
1  2                   China     1,425,671,352  ...       39        65 %     17.72 %
2  3           United States       339,996,563  ...       38        83 %      4.23 %
3  4               Indonesia       277,534,122  ...       30        59 %      3.45 %
4  5                Pakistan       240,485,658  ...       21        35 %      2.99 %

[5 rows x 12 columns]

Process finished with exit code 0
```

Advanced Web Scraping Techniques

Whiles web scraping can be a powerful tool for data collection, it can also be challenging due to several factors. For this write-up, we will focus on 3 common problems; Dynamic content, JavaScript-heavy pages, and Pagination.

Dynamic content refers to web pages that change their content based on user interaction or other factors. This can pose a challenge for web scrapers because the content that needs to be extracted may not be readily available in the initial HTML code. To handle dynamic content, web scrapers can use the Selenium library in Python, or techniques like JavaScript rendering or headless browsers (ZenRows, 2023).

Adding interactivity and dynamic behavior to web sites is common with JavaScript, a scripting language. Pages with a lot of JavaScript might be difficult to scrape since their data may not be easily accessible in the HTML structure. In order to retrieve pertinent data, scrapers must be able to handle the execution of JavaScript. To render JavaScript and scrape the underlying data, Python modules such as Playwright, Selenium, and Scrapy can be utilized (Joyner, 2022).

Most websites contain a huge amount of data, and it is simply not feasible to display all the data on one page. So what do we do? The solution is to show limited records per page and provide access to the remaining records by using pagination (Tamuliunaite, 2021). However, this makes web scraping more complex because the scraper needs to identify and navigate to each page to extract all of the data. This complexity can be overcome with a combination of BeautifulSoup, Requests, Json, and the Urllib libraries in Python.

Using frameworks like Scrapy can quicken the process for complex and large-scale scraping operations. A high-level web crawling and web scraping framework called Scrapy is used to extract structured data from online pages and crawl websites. Additionally, Scrapy offers a reliable architecture for web scraping that makes it possible to manage complicated scraping operations, handle errors, and extract data efficiently. Its features include task scheduling, item pipelines, and data persistence, making it a valuable tool for large-scale scraping projects (Scrapy 2.11 Documentation — Scrapy 2.11.0 Documentation, n.d.)

In some circumstances, APIs (Application Programming Interfaces) can be used instead of web scraping, providing a more direct and regulated approach of collecting data from websites. (What Is an API? - Application Programming Interface Explained - AWS, n.d.). For example, the Weather Channel’s software system contains daily weather data. The weather app on an iPhone “talks” to this system via APIs and shows daily weather updates on the phone.

Additionally, APIs offer standardized data access, frequently with limitations on the amount of data that may be retrieved. Although online scraping may offer greater freedom, using APIs can be a more efficient and compliant way to retrieve structured data. Always keep in mind that responsible online scraping means abiding by the terms of service and data usage rules of the websites being scraped. Furthermore, care should also be taken to avoid flooding websites with queries when scraping since this could lead to performance problems. Both people and companies can successfully use online scraping while abiding by moral standards and honoring the terms of service of websites by carefully evaluating these best practices. (What Is an API? - Application Programming Interface Explained - AWS, n.d.)

Real-world Applications of Web Scraping with Python

Let us now take a look at some real-world applications of web scraping with Python. Several recent studies have highlighted the growing significance of web scraping in various fields. For instance, a study by (Chauhan et al., 2023) emphasizes the role of web scraping in data collection, market research, lead generation, price monitoring, sentiment analysis, content aggregation, academic research, and machine learning.

Data Collection and Market Insights

One of the primary applications of web scraping is data collection. With web scraping, information can be automatically extracted from webpages, revolutionizing data gathering and enabling researchers and organizations to gather enormous datasets for in-depth analysis and insights. (Chauhan et al., 2023). For instance, an e-commerce company can utilize web scraping to collect product data from various online retailers, including product descriptions, specifications, pricing, and customer reviews. This data can be used to perform internal analysis and gain valuable insights into market trends, competitor strategies, and customer preferences.

Lead Generation, Customer Relationship Management, and Dynamic Pricing

Additionally essential to lead creation and customer relationship management (CRM) is web scraping. Web scraping allows businesses to obtain contact details from websites and online directories, which can help with customer database creation, lead nurturing, and CRM strategy improvement (Chauhan et al., 2023). A financial services firm, for example, can employ web scraping to gather information on potential clients, such as their financial profiles, investment preferences, and risk tolerance. This data can then be used to personalize marketing messages, enhance customer service, and build long-term relationships.

Moreover, e-commerce businesses may use web scraping to track rival pricing tactics in real time, giving them the information they need to make competitive pricing decisions and guarantee maximum profit margins (Chauhan et al., 2023). For example, an airline firm can use web scraping to monitor ticket pricing for different routes and dates of travel offered by different carriers. Then, by utilizing this data, pricing strategies can be optimized, income can be maximized, and various consumer segments may be served.

Sentiment Analysis, Brand Reputation Management, and Content Aggregation

Sentiment research and brand reputation management are made easier by web scraping. Through aggregating data from review sites, forums, and social media platforms, businesses can acquire important insights about consumer mood and perceptions of their brands or products. Afterwards, this data can be utilized to strengthen marketing initiatives, resolve consumer complaints, and promote brand reputation (Chauhan et al., 2023). Web scraping can be employed by a cosmetics company, for instance, to collect consumer feedback from social networking platforms, online merchants, and beauty blogs. The cosmetics company may find areas for improvement and drive product development activities by gaining important insights into client sentiment through the analysis of customer reviews from various online sources.

Additionally, by automatically gathering news articles, blog entries, and other online content from a variety of sources, web scraping simplifies the process of content aggregation. Through this technique, news organizations may successfully monitor news updates, identify developing trends, and organize and disseminate information. Web scraping is a technique that news organizations can use to collect news stories, blog posts, and social media content about politics, current affairs, and international affairs. After that, this data can be utilized to choose news feeds, offer thorough coverage, and spot new trends (Chauhan et al., 2023).

Academic Research, Data-Driven Discoveries, and Machine Learning

Web scraping is also important in academic research and data-driven discoveries. Web scraping can be used by researchers to acquire data for a variety of research studies, such as social media analysis, social thought analysis, research data tracking, and data collection for further analysis. This enables data-driven discoveries and breakthroughs in a variety of domains of research (Chauhan et al., 2023). A medical researcher, for example, can use web scraping to collect clinical trial data from a variety of sources, such as pharmaceutical company websites, government databases, and scientific journals. The data collected can then be utilized to assess the efficacy of the treatment, identify potential side effects, and advance medical research.

Lastly, the creation of artificial intelligence algorithms and machine learning models depends heavily on web scraping. Researchers may train these models to predict, categorize, and carry out complicated tasks with surprising precision by gathering massive volumes of data from diverse sources (Chauhan et al., 2023). For instance, a social scientist can gather information on public opinion, online communities, and social media interactions from a variety of platforms, including Facebook, Twitter, and Reddit, by using web scraping techniques. Machine learning models for applications like sentiment analysis, social network analysis, and trend prediction can then be trained using this data.

Connection to written Assignment - DLMDSPWP01

The written assignment aimed to develop a Python program that utilized four training datasets to identify 4 ideal functions out of a pool of 50. The selection criteria involved minimizing the sum of squared y-deviations (Least-Square). The program then evaluated a test dataset using a criterion ensuring the maximum deviation of the calculated regression did not exceed the largest deviation between training data and the chosen ideal function by more than a factor of square root of 2. Based on the code, the chosen functions are ['y3', 'y2', 'y1']. These are the functions selected based on the training data and ideal functions. (All outputs were produced with the PyCharm IDE)

These functions were identified by minimizing the sum of squared y-deviations (Least-Square) for the training data and choosing the function that had the smallest deviation from the ideal functions for each data point. (For reference, the code and resulting output are in Appendix A).

For the implementation, an SQLite database with two tables was compiled; one for training data and another for ideal functions. The test data was then loaded, and if compliant, matched to the chosen ideal functions. The results, including x- and y-values, the corresponding ideal function, and the related deviation, were stored in a new table within the database. Finally, the entire dataset was visualized using the Matplotlib library.

In this case, the average deviation for all 100 test data points is 18.36. This means that on average, the predicted values for the 100 test data points were off by 18.36. This is a large deviation, which suggests that the chosen functions may not be the best fit for all of the data points, and this shows on the visualizations where we see the test data points scattered over the graph, and not in line with the chosen functions.

For reference, the visualizations are in Appendix B).

Web scraping can serve as a valuable tool in enhancing the dataset used for training and testing. By incorporating web scraping techniques (such as the ones we discussed above), we can expand the pool of training datasets, enriching the available data for more comprehensive analysis. This is particularly beneficial when dealing with dynamic datasets that evolve. For example, we had to use different datasets for this assignment, but imagine these datasets were on the IU website and were updated periodically. No problem! With web scraping, we could automate the process of navigating to the site and grabbing the required datasets. We could go further and visualize the trends of chosen ideal functions over time, all automated. That is the power of web scraping.

To move into another industry, if we consider the context of medical research where the aim is to identify optimal treatment patterns, utilizing web scraping techniques will allow us to dynamically fetch the latest clinical trial data, research publications, and patient outcomes. Going further, if we delve deeper into identifying ideal treatments for a specific medical condition, we can use web scraping to continuously update the training datasets with the latest clinical trial results, ensuring a comprehensive set of observations. Moreover, this opens avenues to discover new ideal functions that may represent emerging and effective treatment approaches.

Conclusion

In conclusion, the topic "Extracting Insights from the Web: Web Scraping with Python" was selected due to the growing importance of web scraping in handling the abundance of online information. Throughout this paper, we provided a clear understanding of web scraping, considering ethical and legal aspects. We also covered the fundamentals, introduced essential Python libraries for scraping web data, delved a bit into advanced techniques, and showcased real-world applications.

To summarize the points, it is clear that web scraping may be a valuable tool for making educated decisions when done carefully and morally. In future, questions regarding the ever-changing landscape of web scraping and its possible effects on data privacy will likely open up new directions for this constantly developing field of study.

References

Amos, D. (2023, February 24). *A Practical Introduction to Web Scraping in Python*. Real Python. Retrieved December 14, 2023, from https://realpython.com/python-web-scraping-practical-introduction/

Chauhan, R., Negi, A., & Manchanda, M. (2023). An Extensive Review on Web Scraping Technique using Python. *2023 Second International Conference on Augmented Intelligence and Sustainable Systems (ICAISS)*. https://doi.org/10.1109/icaiss58487.2023.10250745

Dahito, M. (2023, November 3). Web Scraping 101: A Beginners Guide. *Ubique Digital Solutions*. Retrieved December 14, 2023, from https://ubiquedigitalsolutions.com/blog/web-scraping-101-a-beginners-guide/

Infotech, H. (2022, October 7). *7 Reasons Why Web Scraping is Popular Currently*. https://www.linkedin.com/pulse/7-reasons-why-web-scraping-popular-currently-hir-infotech/

Joyner, J. (2022, April 9). How to scrape JavaScript heavy sites like a pro with Python. *Medium*. Retrieved December 13, 2023, from https://medium.com/thedevproject/how-to-scrape-javascript-heavy-sites-like-a-pro-with-python-1ecf6f829538

Odier, G. (2018, November 19). Web scraping. *11 reasons why you should use web scraping*. Retrieved December 18, 2023, from https://www.captaindata.co/blog/11-reasons-why-use-web-scraping

Ozsahan, H. (2023, December 21). *15 Best Web scraping tools in 2024 to extract online data*. Popupsmart. Retrieved December 13, 2023, from https://popupsmart.com/blog/web-scraping-tools

pandas documentation — pandas 2.1.4 documentation. (2023, December 8). Retrieved December 15, 2023, from https://pandas.pydata.org/docs/index.html

Richardson, L. (2023). *Beautiful Soup Documentation — Beautiful Soup 4.4.0 documentation*. Read the Docs. Retrieved December 15, 2023, from https://beautiful-soup-4.readthedocs.io/en/latest/

Scrapy 2.11 documentation — Scrapy 2.11.0 documentation. (n.d.). Retrieved December 13, 2023, from https://docs.scrapy.org/en/latest/

Statista. (2023, November 27). *Web analytics software market share worldwide 2023*. https://www.statista.com/statistics/1258557/web-analytics-market-share-technology-worldwide/

Tamuliunaite, V. (2021, July 6). *Pagination in web scraping*. Retrieved December 13, 2023, from https://oxylabs.io/blog/pagination-in-web-scraping

Webscrape AI. (2023, May 11). Webscrape AI. *What is Web Scraping? A Quick Overview for Beginners*. Retrieved December 13, 2023, from https://webscrapeai.com/blog/what-is-web-scraping-a-quick-overview-for-beginners

What is an API? - Application Programming Interface Explained - AWS. (n.d.). Amazon Web Services, Inc. Retrieved December 13, 2023, from https://aws.amazon.com/what-is/api/

ZenRows. (2023, October 14). Dynamic Web Pages Scraping with Python: Guide to Scrape All Content. *ZenRows*. Retrieved December 13, 2023, from https://www.zenrows.com/blog/dynamic-web-pages-scraping-python#conclusion

Appendix A.

Python code for Written Assignment

```python
# Here we import the necessary libraries for data processing, visualization, and
testing
import pandas as pd
import matplotlib.pyplot as plt
from sqlalchemy import create_engine, Table, Column, Float, MetaData, String, inspect
from sklearn.linear_model import LinearRegression
import numpy as np
import unittest


# Here we define a function to create the necessary tables in the database. Since we
are using SQLAlchemy, we use MetaData.
# This allows us to store information about the database structures we will use.
def create_db(database_engine):
    metadata = MetaData()
    # Here we define our tables for training data, ideal functions, and results
    Table('training_data', metadata, Column('x', Float), Column('y1', Float),
Column('y2', Float),
          Column('y3', Float), Column('y4', Float))
    Table('ideal_functions', metadata, Column('x', Float), Column('y1', Float),
Column('y2', Float),
          Column('y3', Float), Column('y4', Float))
    Table('results', metadata, Column('x', Float), Column('y', Float),
Column('chosen_function', String),
          Column('deviation', Float))
    # We can now create the tables in the database
    metadata.create_all(database_engine)

# Now we can define a function to load the training data from the CSV file into the
database we have created. We incorporate error handling with our code
def load_training(path, database_engine):
    try:
        training_data = pd.read_csv(path)
    except FileNotFoundError:
        raise FileNotFoundError(f"File '{path}' not found.")
    training_data.to_sql('training_data', con=database_engine, if_exists='replace',
index=False)


# Next we define a function to load the ideal functions from the CSV file into the
database
def load_ideal(path, database_engine):
    try:
        ideal_functions = pd.read_csv(path)
    except FileNotFoundError:
        raise FileNotFoundError(f"File '{path}' not found.")
    ideal_functions.to_sql('ideal_functions', con=database_engine,
if_exists='replace', index=False)

# Here we first define a function to choose the best-fit functions based on the
training data and ideal functions
def choose_funcs(database_engine):
    chosen_functions = select_best_fit_funcs(database_engine)
    return chosen_functions

# According to the defined criteria, we then define another function to select the
four best-fit functions based on the training data and ideal functions.
def select_best_fit_funcs(database_engine):
    funcs = []
    training_data = pd.read_sql('training_data', con=database_engine)
    ideal_functions = pd.read_sql('ideal_functions', con=database_engine)

    training_x = training_data['x'].values
    for f in ideal_functions.columns[1:]:
        if f not in training_data.columns:
            continue
        ideal_y = ideal_functions[f].values
        m = LinearRegression().fit(training_x.reshape(-1, 1), training_data[f])
        y_pred = m.predict(training_x.reshape(-1, 1))
        deviations = (training_data[f] - y_pred) ** 2
        ssd = deviations.sum()
        if not funcs or ssd < min([func[1] for func in funcs]):
            funcs.append((f, ssd))
    funcs.sort(key=lambda x: x[1])
    return [f[0] for f in funcs[:4]]

# We then define a function to match our test data to the chosen functions and save
the results
def match_test(path, database_engine):
    # Choose functions
    chosen_functions = choose_funcs(database_engine)

    # This is to check if the chosen_functions is empty
    if not chosen_functions:
        print("No functions chosen. Exiting match_test.")
        return

    results = pd.DataFrame(columns=['x', 'y', 'chosen_function', 'deviation'])
    test_data = pd.read_csv(path)
    for _, r in test_data.iterrows():
        x = r['x']
        chosen_f = match_to_chosen_func(x, chosen_functions, database_engine)
        if chosen_f:
            deviation = calculate_deviation(r['y'], chosen_f, database_engine)
            results = pd.concat([results, pd.DataFrame(
                {'x': [x], 'y': [r['y']], 'chosen_function': [chosen_f], 'deviation':
[deviation]})],
                                ignore_index=True)

# We need to convert the deviation column values into numeric format
    results['deviation'] = results['deviation'].astype(float)

    results.to_sql('results', con=database_engine, if_exists='replace', index=False)
    results.to_sql('test_results', con=database_engine, if_exists='replace',
index=False)

    # Create visualizations
    plot_ideal_functions(chosen_functions, database_engine)
    plot_scatter_test_data(chosen_functions, results, database_engine)


# Now we define a function to plot the ideal functions and the chosen function
def plot_ideal_functions(chosen_functions, database_engine):
    ideal_functions = pd.read_sql('ideal_functions', con=database_engine)
    for chosen_f in chosen_functions:
        plt.plot(ideal_functions['x'], ideal_functions[chosen_f], label=chosen_f,
linestyle='solid')

    # For visualization purposes, we plot the chosen function as a smaller line
    chosen_function = chosen_functions[0]
    plt.plot(ideal_functions['x'], ideal_functions[chosen_function],
             label=f"Chosen Function ({chosen_function})", color='blue',
linestyle='dashed')

    plt.xlabel("x")
    plt.ylabel("y")
    plt.legend()
    plt.title("Ideal Functions and Chosen Functions")
    plt.show()

# Then we define a function to plot the test data and the chosen function
def plot_scatter_test_data(chosen_functions, results, database_engine):
    ideal_functions = pd.read_sql('ideal_functions', con=database_engine)
    for chosen_f in chosen_functions:
        plt.plot(ideal_functions['x'], ideal_functions[chosen_f], label=chosen_f,
linestyle='solid')

    chosen_function = chosen_functions[0]
    plt.plot(ideal_functions['x'], ideal_functions[chosen_function],
             label=f"Chosen Function ({chosen_function})", color='blue',
linestyle='dashed')

# For visualization purposes, we change the color of the test data points to red
    plt.scatter(results[results['chosen_function'] == chosen_function]['x'],
                results[results['chosen_function'] == chosen_function]['y'],
                label="Test Data", color='red', s=10)
    plt.xlabel("x")
    plt.ylabel("y")
    plt.legend()
    plt.title("Test Data vs. Chosen Function")
    plt.show()

# Then we define another function to match a test data point to a chosen function
def match_to_chosen_func(x, chosen_functions, database_engine):
    if not chosen_functions:
        return None
    ideal_functions = pd.read_sql('ideal_functions', con=database_engine)
    ideal_x, ideal_y = ideal_functions['x'].values,
ideal_functions[chosen_functions].values
    diffs = (ideal_x - x) ** 2
    idx = np.argmin(diffs)
    if idx < len(chosen_functions):
        chosen_f = chosen_functions[idx]
    else:
        chosen_f = chosen_functions[0]
    return chosen_f

# Here we define a function to calculate the deviation between the test data and
chosen function
def calculate_deviation(y, chosen_f, database_engine):
    ideal_y = pd.read_sql_query(f"SELECT {chosen_f} FROM ideal_functions;",
database_engine)[chosen_f].values
    deviation = np.abs(y - ideal_y)

    if deviation.size == 1:
        return float(deviation)
    else:
        return float(np.mean(deviation))

# The code below is the main section for running the whole script
# We need to define the database connection string
db_connection_string = 'sqlite:///test_data.db'
engine = create_engine(db_connection_string)

# After, we create the necessary tables in the database
```

```
create_db(engine)

# Then we load the training data
training_data_path = r"C:\Users\Noel\Documents\IU International\Course
Materials\Programming with Python\Written Exam\dataset\train.csv"
load_training(training_data_path, engine)

# Now we load the ideal functions
ideal_functions_path = r"C:\Users\Noel\Documents\IU International\Course
Materials\Programming with Python\Written Exam\dataset\ideal.csv"
load_ideal(ideal_functions_path, engine)

# We then choose the functions
chosen_functions = choose_funcs(engine)

# Then we proceed to match the test data
test_data_path = r"C:\Users\Noel\Documents\IU International\Course
Materials\Programming with Python\Written Exam\dataset\test.csv"
match_test(test_data_path, engine)

# We finally print the chosen functions
print(chosen_functions)

# This line of code is to get information about the tables
inspector = inspect(engine)

# This is to get a list of all tables in the database
table_names = inspector.get_table_names()

# We can then print the contents of each table. We use a ‘for loop’ to iterate through
The list of table names, print the content. The last line of code is for better
readability
for table_name in table_names:
    print(f"Table: {table_name}")
    query = f"SELECT * FROM {table_name};"
    table_data = pd.read_sql_query(query, engine)
    print(table_data)
    print("\n" + "=" * 50 + "\n")

# We want the average deviation, so we first need to load the 'results' table
results = pd.read_sql('results', con=engine)

# Then proceed to calculate the average deviation
average_deviation = results['deviation'].mean()

print(f"Average Deviation: {average_deviation}")
```

Output from Code

```
 ['y3', 'y2', 'y1']

Table: ideal_functions

        x        y1        y2  ...       y48       y49       y50

0   -20.0 -0.912945  0.408082  ... -0.186278  0.912945  0.396850

1   -19.9 -0.867644  0.497186  ... -0.215690  0.867644  0.476954

2   -19.8 -0.813674  0.581322  ... -0.236503  0.813674  0.549129

3   -19.7 -0.751573  0.659649  ... -0.247887  0.751573  0.612840

4   -19.6 -0.681964  0.731386  ... -0.249389  0.681964  0.667902

..    ...       ...       ...  ...       ...       ...       ...

395  19.5  0.605540  0.795815  ...  0.240949  0.605540  0.714434

396  19.6  0.681964  0.731386  ...  0.249389  0.681964  0.667902

397  19.7  0.751573  0.659649  ...  0.247887  0.751573  0.612840

398  19.8  0.813674  0.581322  ...  0.236503  0.813674  0.549129

399  19.9  0.867644  0.497186  ...  0.215690  0.867644  0.476954

[400 rows x 51 columns]


==================================================

Table: results

       x          y chosen_function  deviation

0   17.5  34.161040              y3  24.163322

1    0.3   1.215102              y3   8.782615

2   -8.7 -16.843908              y3  26.841626

3  -19.2 -37.170870              y3  47.168588
```

```
4  -11.0 -20.263054              y3  30.260772

..   ...        ...            ...        ...

95  -1.9  -4.036904              y3  14.034622

96  12.2  -0.010358              y3  10.008076

97  16.5 -33.964134              y3  43.961852

98   5.3 -10.291622              y3  20.289340

99  17.9  28.078455              y3  18.080737

[100 rows x 4 columns]
```

Table: test_results

```
       x          y chosen_function  deviation

0   17.5  34.161040              y3  24.163322

1    0.3   1.215102              y3   8.782615

2   -8.7 -16.843908              y3  26.841626

3  -19.2 -37.170870              y3  47.168588

4  -11.0 -20.263054              y3  30.260772

..   ...        ...            ...        ...

95  -1.9  -4.036904              y3  14.034622

96  12.2  -0.010358              y3  10.008076

97  16.5 -33.964134              y3  43.961852

98   5.3 -10.291622              y3  20.289340

99  17.9  28.078455              y3  18.080737

[100 rows x 4 columns]
```

```
==================================================

Table: training_data

        x        y1        y2        y3        y4

0   -20.0  39.778572 -40.078590 -20.214268 -0.324914

1   -19.9  39.604813 -39.784000 -20.070950 -0.058820

2   -19.8  40.099070 -40.018845 -19.906782 -0.451830

3   -19.7  40.151100 -39.518402 -19.389118 -0.612044

4   -19.6  39.795662 -39.360065 -19.815890 -0.306076

..    ...        ...        ...        ...       ...

395  19.5 -38.254158  39.661987  19.536741  0.695158

396  19.6 -39.106945  39.067880  19.840752  0.638423

397  19.7 -38.926495  40.211475  19.516634  0.109105

398  19.8 -39.276672  40.038870  19.377943  0.189025

399  19.9 -39.724934  40.558865  19.630678  0.513824

[400 rows x 5 columns]

==================================================

Average Deviation: 18.3649951950925

Process finished with exit code 0
```

Appendix B.

Visualizations for Written Assignment

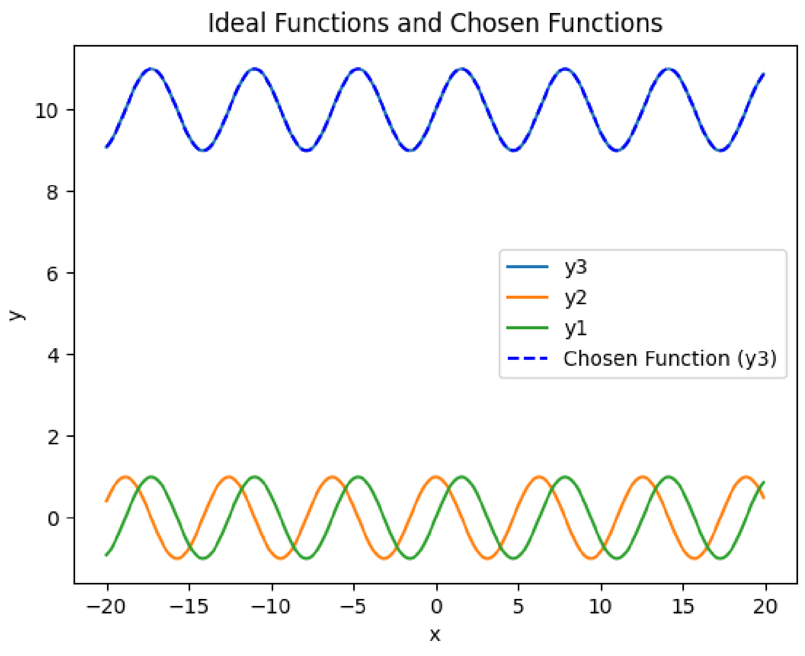


Image 1: Ideal Functions and Chosen Functions

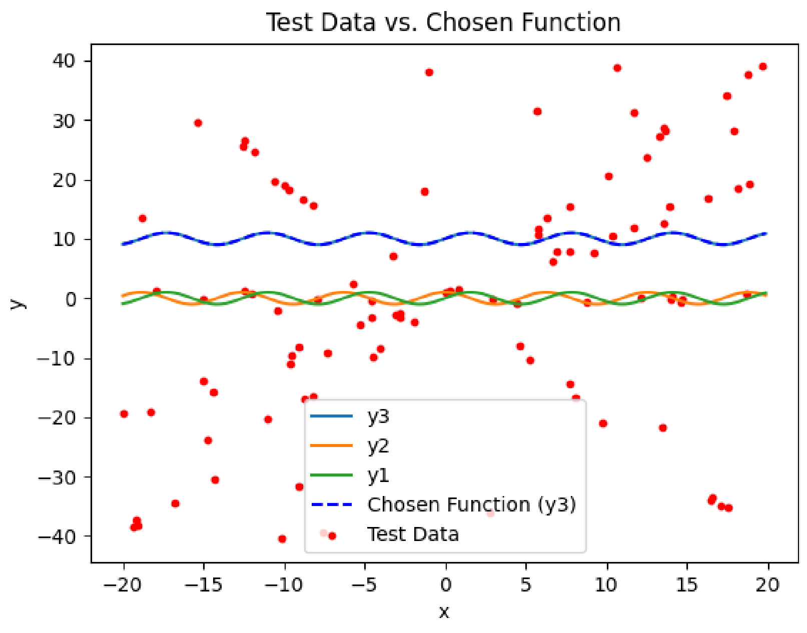


Image 2: Test Data vs. Chosen Function