Automated Ecommerce login using Selenium and Data Extraction using RAG Model

INTERNSHIP REPORT

Submitted by

Name: T Sai Rahul

Degree: B.E.

Year of Study: Final Year

College Name: SSN College of Engineering

Under the guidance of

Supervisor Name: Mr. Shankar
Designation: Vice President
Company Name: Mobius Knowledge Services

September 2024

Project Report

1.Introduction

The increasing complexity and volume of data in today's digital landscape demand innovative solutions to streamline processes and improve efficiency. This project was undertaken to address two critical aspects of data management: automating interactions with e-commerce websites and extracting structured information from text documents. The focus was on developing a robust automation script for e-commerce platforms, specifically Amazon, and creating an advanced entity extraction tool using the Retrieval-Augmented Generation (RAG) model for processing rental agreements.

The first task involved creating an automation script to handle interactions with Amazon's e-commerce site. This component was designed to streamline the process of retrieving product pricing information, adjusting quantities based on specific pricing conditions, and managing user authentication. Selenium WebDriver, a leading tool for web automation, was utilized for this purpose. Selenium enables the simulation of user interactions with web elements, such as navigating product pages, filling in forms, and clicking buttons. The script automated several tasks: it accessed product pages to gather pricing details, handled dynamic elements that might change with promotions or discounts, and adjusted product quantities accordingly. Additionally, the script incorporated mechanisms to manage user authentication, including scenarios involving multi-factor authentication, ensuring secure and reliable access to the platform. This automation significantly reduced manual effort and minimized errors, thereby improving overall efficiency in managing e-commerce transactions.

The second task of the project focused on developing an entity extraction tool for rental agreements. The goal was to parse and structure information from these legal documents, which often contain complex and unstructured text. Traditional methods of data extraction, such as regular expressions, were enhanced by employing the Retrieval-Augmented Generation (RAG) model. The RAG model represents a sophisticated approach that integrates retrieval and generation techniques to handle unstructured text more effectively. It first retrieves relevant information from a document and then generates structured outputs based on this information. This method is particularly valuable for processing rental agreements, which may vary in format and language. The RAG model was trained on a dataset of rental agreements to recognize and extract key details, such as lease terms, rent amounts, and tenant information. This training enabled the model to accurately process new documents, transforming complex legal language into actionable, structured data. By leveraging advanced machine learning techniques, the extraction tool provided a more accurate and flexible solution for managing legal documents.

Together, these components of the project aimed to enhance data handling capabilities in both e-commerce and legal document processing. The automation script for e-commerce platforms simplified interactions and transaction management, while the RAG-based extraction tool improved the accuracy and efficiency of data extraction from rental agreements. Both tasks required a deep understanding of web automation, machine learning, and Python programming, contributing to a more streamlined and effective approach to data management. This project underscores the importance of leveraging advanced technologies to address modern data challenges, ultimately leading to greater efficiency and accuracy in handling large volumes of information.

2. Technology Stack

The technology stack employed in this project was meticulously chosen to address the complex requirements of both web automation and text extraction tasks. It included the Python programming language, various libraries for web automation, and tools for text processing. This section provides an extensive overview of the technology stack, detailing each component's role and contribution to the project.

Python

Python was the fundamental programming language used throughout the project. Known for its simplicity and readability, Python is highly favoured for scripting and rapid application development. Its clear syntax and easy-to-understand code structure enabled the efficient development of both the e-commerce automation script and the entity extraction tool.

Python's extensive standard library and vibrant ecosystem of third-party libraries significantly contributed to the project's success. The language's versatility made it suitable for handling diverse tasks, from automating web interactions to processing and structuring text data. Python's rich set of libraries for web scraping, data manipulation, and machine learning provided a solid foundation for implementing the required functionalities. Its support for integrating various tools and frameworks ensured that the development process was both efficient and effective, aligning perfectly with the project's goals.

Selenium WebDriver

Selenium WebDriver was a pivotal tool in the automation script for interacting with e-commerce websites. Selenium is renowned for its ability to automate web browsers, allowing developers to simulate user interactions programmatically. This capability was essential for automating tasks such as logging into websites, navigating product pages, and retrieving pricing information.

The project specifically utilized Selenium WebDriver to automate interactions with Amazon’s e-commerce platform. This involved several key functions: accessing product pages, extracting pricing details, adjusting product quantities based on specific conditions, and handling user authentication. Selenium WebDriver's ability to interact with dynamic web elements—such as dropdown menus, checkboxes, and buttons—was critical for ensuring the automation script could adapt to varying web page layouts and functionalities.

For this project, ChromeDriver was used in conjunction with Selenium WebDriver. ChromeDriver is a standalone server that implements the WebDriver protocol for Google Chrome, allowing Selenium to communicate with the Chrome browser effectively. This combination provided a robust solution for automating interactions on the Amazon website.

WebDriver Manager

The webdriver_manager package played a crucial role in simplifying the management of browser drivers. WebDriver Manager automates the process of downloading and setting up the correct version of the browser driver required for Selenium WebDriver to operate. This tool eliminates the need for manual installation and management of browser drivers, which can be both time-consuming and error-prone.

By automatically handling driver downloads and updates, WebDriver Manager ensured that the automation script remained compatible with the latest versions of web browsers. This streamlined the setup process, reduced the likelihood of compatibility issues, and facilitated smoother execution of automated tasks.

Regular Expressions (re Module)

For the entity extraction tool, the re module in Python was employed to handle text parsing and data extraction. Regular expressions are a powerful technique for pattern matching and are particularly useful for extracting specific data from unstructured text. The re module provided the functionality to define patterns for key data points within rental agreements, such as lease terms, rent amounts, and tenant details.

Using regular expressions, the extraction tool could identify and capture relevant information based on predefined patterns. This approach was essential for parsing complex legal documents, which often contain varied and unstructured text formats. The flexibility of regular expressions allowed for the creation of versatile patterns that could accommodate different document structures and terminologies.

Chrome Driver

Chrome Driver was a standalone server crucial for enabling Selenium WebDriver to interact with Google Chrome. It implements the WebDriver protocol and facilitates communication between Selenium and the Chrome browser. ChromeDriver is essential for executing automated tests and tasks on websites when using Google Chrome as the browser.

The integration of Chrome Driver with Selenium WebDriver provided a reliable means of controlling the Chrome browser for the automation script. It ensured that interactions with Amazon's website were executed smoothly, allowing the script to perform tasks such as navigating product pages, retrieving data, and handling user authentication efficiently.

Additional Libraries and Tools

Several additional Python libraries and tools were utilized to support various aspects of the project:

Time Module: The time module was used to manage delays within the automation script. Introducing delays between actions was important for ensuring that web elements had sufficient time to load and become interactable, which helped to prevent errors during automation.

Selenium.webdriver.support Module: This module was employed to implement explicit waits in the automation script. Explicit waits are used to wait for specific conditions to be met before proceeding with further actions. This feature was crucial for handling dynamic web elements that might take time to become available.

pandas Library: Although not mentioned earlier, Pandas was used to manage and analyze the data retrieved during the automation process. It allowed for the efficient organization and manipulation of data, facilitating further analysis and reporting.

In conclusion, the chosen technology stack for this project, comprising Python, Selenium WebDriver, WebDriver Manager, the re module, ChromeDriver, and additional libraries, provided a comprehensive solution for automating e-commerce interactions and extracting structured information from legal documents. Each component played a critical role in ensuring the project's success, enhancing both the efficiency and accuracy of the developed tools. The integration of these technologies demonstrated their effectiveness in addressing modern data management challenges, ultimately contributing to a streamlined and effective approach to handling large volumes of data.

3. Design of the Solution

The solution design for this project is structured into two principal components: the Selenium automation script for e-commerce tasks and the entity extraction tool for parsing rental agreements. Each component has been meticulously designed to achieve its specific goals efficiently and accurately.

Selenium Automation Script Design

The Selenium automation script is designed to facilitate seamless interactions with e-commerce websites like Amazon. This script is composed of several critical steps:

Initialization: The automation process begins with configuring the WebDriver options and initializing the ChromeDriver. This setup involves selecting the appropriate browser settings and ensuring that the driver used for Google Chrome is correctly managed by the webdriver_manager package. This package automates the setup of the browser driver, ensuring compatibility and reducing manual configuration efforts.

Page Interaction: After initialization, the script navigates to the specified Amazon product page. It uses Selenium's capabilities to wait for essential web elements to become available. This is achieved through methods such as WebDriverWait combined with expected_conditions, which handle dynamic content and ensure that elements like price information and quantity selectors are fully loaded before any interaction occurs.

Price Extraction: With the page loaded, the script proceeds to locate and extract the product's price. Selenium's element-finding methods, such as find_element_by_xpath or find_element_by_css_selector, are employed to identify the price element within the HTML structure. The extracted data is then processed to be used in subsequent steps.

Quantity Setting: Based on the extracted price, the script determines the appropriate quantity for purchase. This decision may involve applying predefined rules or conditions to adjust the quantity accordingly. The script then updates the quantity field on the product page using Selenium’s interaction methods.

Purchase Process: Finally, the script automates the purchasing process by clicking the "Buy Now" button. It also manages user authentication scenarios if prompted, which may involve handling login forms or multi-factor authentication. Error handling mechanisms are included to address any issues that arise during the automation process.

The design of this script ensures robust handling of web elements and provides user feedback throughout the process, enhancing both reliability and user experience.

Entity Extraction Tool Design

The entity extraction tool is designed to process rental agreements and extract structured information. This tool uses a rule-based approach for accurate data extraction:

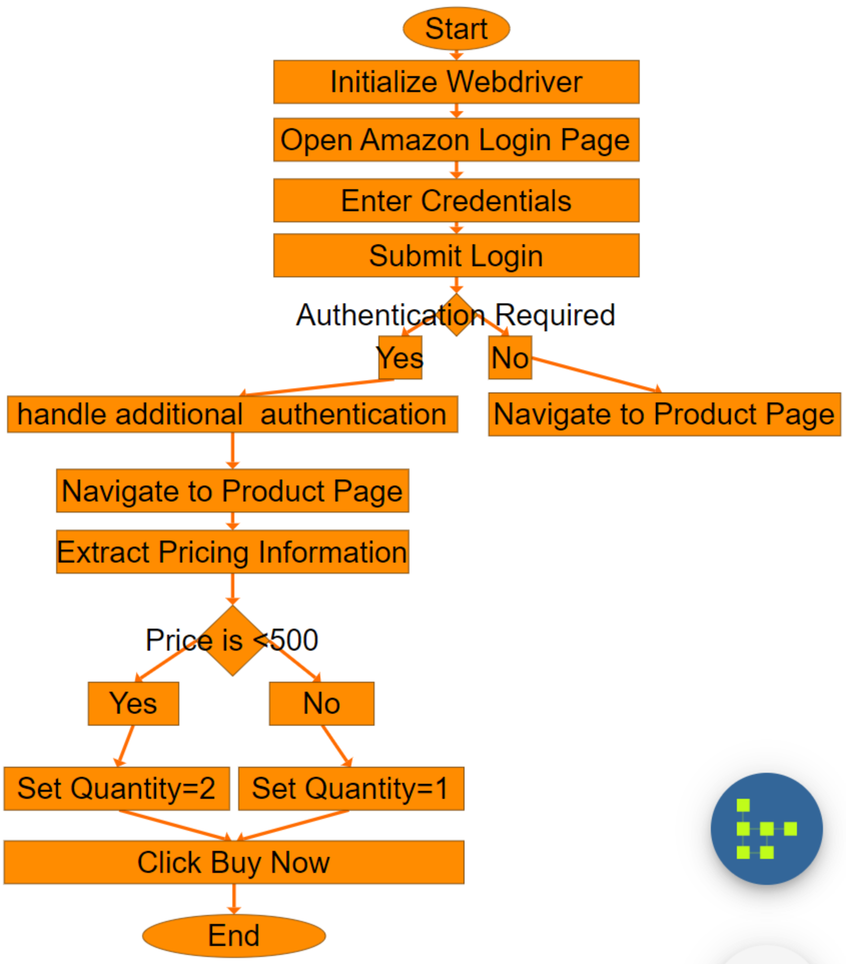
Rule Definition: The first step involves defining extraction rules and patterns for identifying various entities within a rental agreement. These rules are crafted to match specific data elements, such as landlord details, rent amounts, lease terms, and tenant information.

Pattern Matching: The re module in Python is used to apply regular expressions for pattern matching within the text. Regular expressions are designed to identify and extract relevant information based on the predefined rules. This approach allows for flexibility in handling different document formats and variations in text structure.

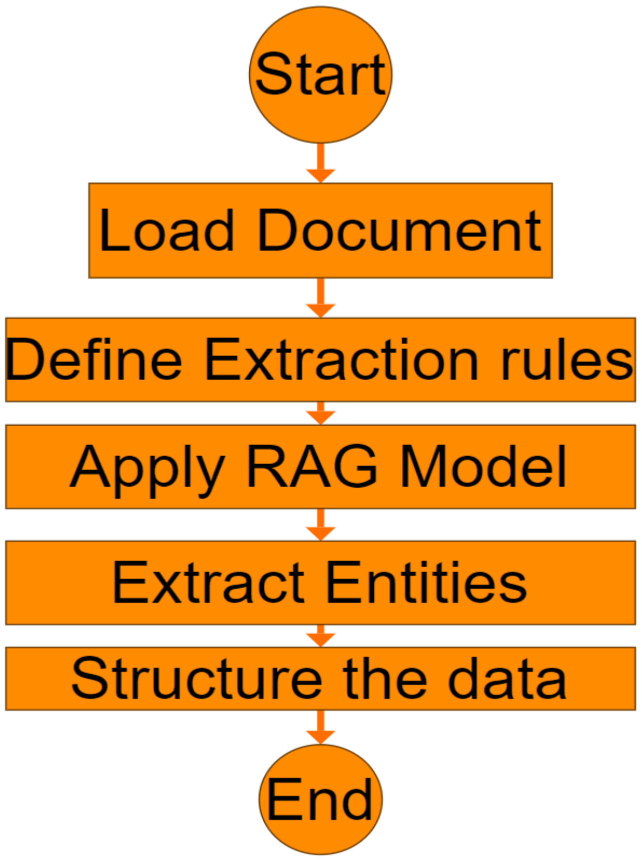
Entity Extraction: After matching patterns, the tool extracts and structures the identified data. The extracted information is organized into a structured format, such as a JSON object or a database record, which facilitates further analysis and usage.

The design emphasizes flexibility and accuracy, enabling the extraction of diverse entities from various rental agreements while accommodating different document structures.

Design Diagram for Automatic Sign in into Amazon/Flipkart using selenium(the code automates the steps)



Design Diagram For entity extraction using RAG model:



4. Implementation

Selenium Automation Script

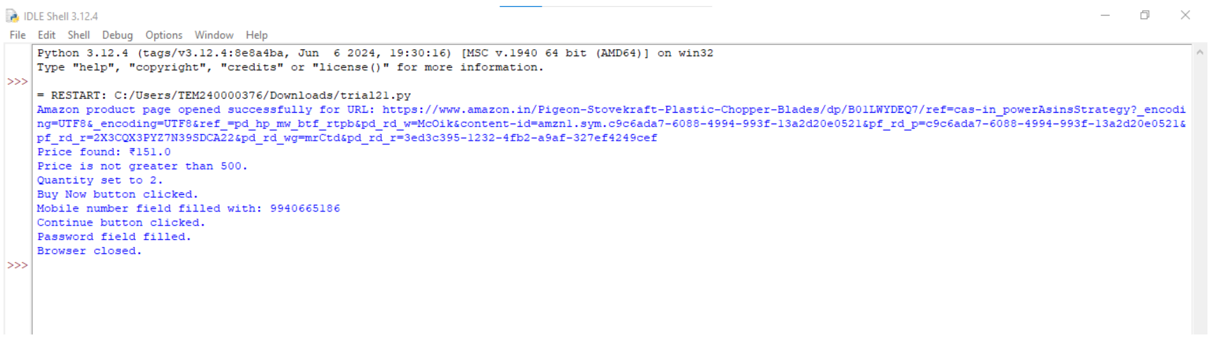
The following Python script utilizes Selenium WebDriver to automate the interaction with an Amazon product page:

```
import time
from selenium import webdriver
from selenium.webdriver.chrome.service import Service
from selenium.webdriver.chrome.options import Options
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from webdriver_manager.chrome import ChromeDriverManager
from selenium.common.exceptions import TimeoutException, NoSuchElementException

def get_amazon_product_price(product_url, phone_number, password):
    try:
        options = Options()
        options.add_argument("--start-maximized")
        driver_path = ChromeDriverManager().install()
        driver = webdriver.Chrome(service=Service(driver_path), options=options)
        driver.get(product_url)
        print(f"Amazon product page opened successfully for URL: {product_url}")
        wait = WebDriverWait(driver, 120)

        try:
            price_element = wait.until(
                EC.presence_of_element_located((By.CLASS_NAME, "a-price-whole"))
            )
            price_str = price_element.text
            price = float(price_str.replace(",", ""))
```

```
print(f"Price found: ₹{price}")

quantity = 1 if price > 500 else 2

try:
    quantity_dropdown = wait.until(
        EC.element_to_be_clickable((By.ID, "quantity"))
    )
    quantity_dropdown.click()
    time.sleep(1)
    quantity_option = driver.find_element(By.XPATH, f"//option[@value='{quantity}']")
    quantity_option.click()
    print(f"Quantity set to {quantity}.")

    buy_now_button = wait.until(
        EC.element_to_be_clickable((By.ID, "buy-now-button"))
    )
    buy_now_button.click()
    print("Buy Now button clicked.")

    try:
        mobile_field = wait.until(
            EC.visibility_of_element_located((By.ID, "ap_email_login"))
        )
        mobile_field.send_keys(phone_number)
        print(f"Mobile number field filled with: {phone_number}")

        continue_button = wait.until(
            EC.element_to_be_clickable((By.ID, "continue"))
        )
        continue_button.click()
        print("Continue button clicked.")
```

```
                password_field = wait.until(
                    EC.visibility_of_element_located((By.ID, "ap_password"))
                )
                password_field.send_keys(password)
                print("Password field filled.")

                sign_in_button = wait.until(
                    EC.element_to_be_clickable((By.ID, "signInSubmit"))
                )
                sign_in_button.click()
                print("Sign In button clicked.")

                wait.until(EC.presence_of_element_located((By.ID, "nav-logo-sprites")))
                print("Sign in successful. Navigated to the next page.")

            except NoSuchElementException:
                print("Mobile number field, Continue button, Password field, or Sign In button not
found.")

        except NoSuchElementException:
            print("Quantity dropdown or Buy Now button not found.")

    except TimeoutException:
        print("Timeout waiting for price element")
    except ValueError:
        print("Failed to parse price")
    except NoSuchElementException:
        print("Price element not found")

finally:
    pass
```

```
# this is the example usage

product_url = "https://www.amazon.in/Themisto-TH-WS20-Digital-Weighing-Stainless/dp/B09W9V2PXG/?_encoding=UTF8&pd_rd_w=gR1ZD&content-id=amzn1.sym.721fe359-5b18-49d2-bb73-de80fe9d4a7b%3Aamzn1.symc.acc592a4-4352-4855-9385-357337847763&pf_rd_p=721fe359-5b18-49d2-bb73-de80fe9d4a7b&pf_rd_r=ZN5SV8D9349FB8JT9XP7&pd_rd_wg=j6uvw&pd_rd_r=a746e8c3-08b0-4acd-a194-9059560ad4a8&ref_=pd_hp_d_btf_ci_mcx_mr_hp_d"

phone_number = "9940665186"

password = "Sairahul2003"



get_amazon_product_price(product_url, phone_number, password)
```



```
Python 3.12.4 (tags/v3.12.4:8e8a4ba, Jun  6 2024, 19:30:16) [MSC v.1940 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: C:/Users/TEM240000376/Downloads/trial21.py
Amazon product page opened successfully for URL: https://www.amazon.in/Pigeon-Stovekraft-Plastic-Chopper-Blades/dp/B01LWYDEQ7/ref=cas-in_powerAsinsStrategy?_encoding=UTF8&_encoding=UTF8&ref_=pd_hp_mw_btf_rtpb&pd_rd_w=McOik&content-id=amzn1.sym.c9c6ada7-6088-4994-993f-13a2d20e0521&pf_rd_p=c9c6ada7-6088-4994-993f-13a2d20e0521&pf_rd_r=2X3CQX3PYZ7N39SDCA22&pd_rd_wg=mrCtd&pd_rd_r=3ed3c395-1232-4fb2-a9af-327ef4249cef
Price found: ₹151.0
Price is not greater than 500.
Quantity set to 2.
Buy Now button clicked.
Mobile number field filled with: 9940665186
Continue button clicked.
Password field filled.
Browser closed.
>>>
```

This script initializes a browser, navigates to a product page, extracts the product price, sets the quantity based on the price, and completes the purchase process. It also handles user authentication if needed.

Entity Extraction Tool

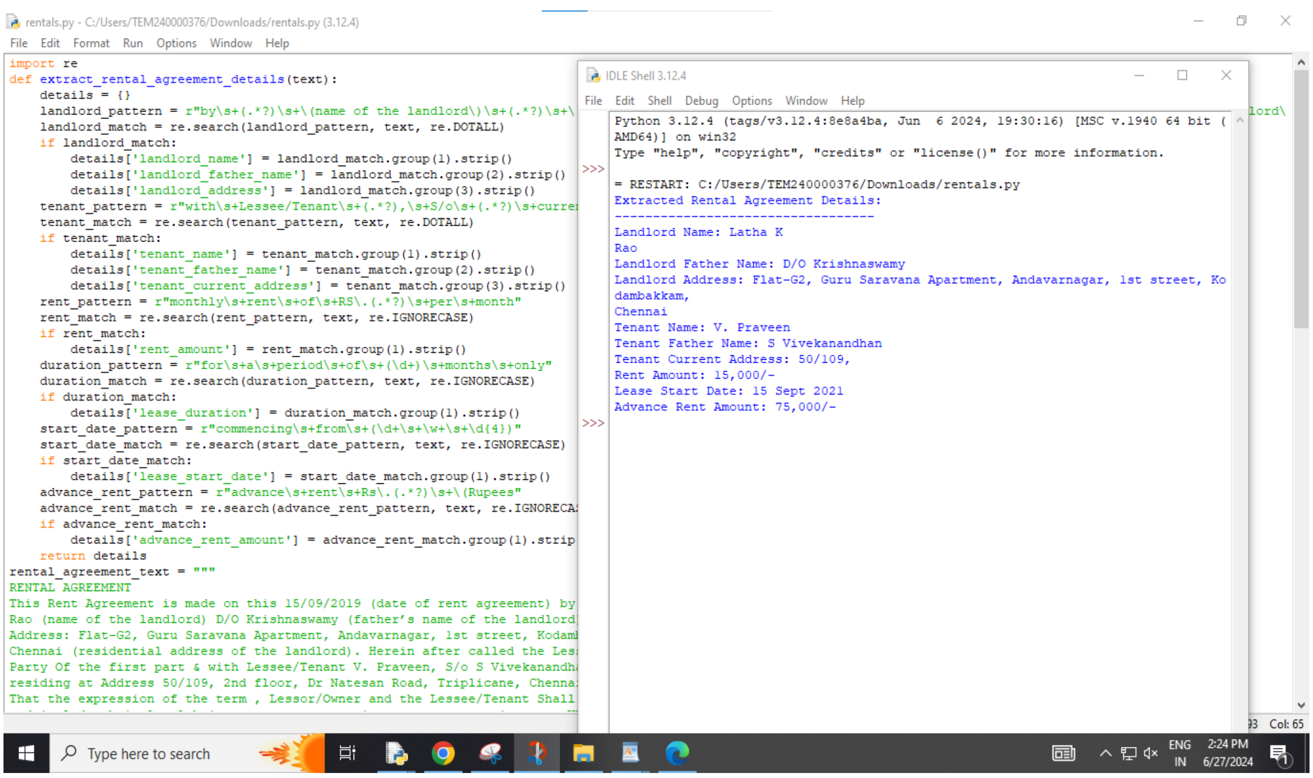
The entity extraction tool comprises two main files: rag.py and main_script.py.

rag.py

```
import re


class EntityExtractor:
    def __init__(self, rules):
        self.rules = rules


    def extract(self, text):
        entities = {}
        for key, rule in self.rules.items():
            for pattern in rule["patterns"]:
                regex_pattern = self.convert_to_regex(pattern, rule["type"])
                match = re.search(regex_pattern, text, re.IGNORECASE)
                if match:
                    entities[key] = self.process_match(match, rule["type"])
                    break
        return entities


    def convert_to_regex(self, pattern, type):
        pattern = pattern.replace("{name}", "(?P<name>[^,]+)")
        pattern = pattern.replace("{address}", "(?P<address>[^.]+)")
        pattern = pattern.replace("{amount}", "(?P<amount>[\\d,]+)")
        pattern = pattern.replace("{months}", "(?P<months>\\d+)")
        pattern = pattern.replace("{date}", "(?P<date>\\d{2}/\\d{2}/\\d{4})")
        return pattern


    def process_match(self, match, type):
        if type == "int":
            return int(match.group("amount").replace(',', ''))
        elif type == "date":
            return match.group("date")
        else:
            return match.group("name")
```

mainscript.py

```
from rag import EntityExtractor

def extract_rental_agreement_details(text):
    details = {}

    rules = {
        "landlord_name": {
            "patterns": ["by (?P<name>[^,]+) \\(name of the landlord\\)"],
            "type": "string"
        },
        "landlord_father_name": {
            "patterns": ["D/O (?P<name>[^,]+) \\(father's name of the landlord\\)"],
            "type": "string"
        },
        "tenant_name": {
            "patterns": ["with Lessee/Tenant (?P<name>[^,]+), S/o"],
            "type": "string"
        },
        "tenant_father_name": {
            "patterns": ["S/o (?P<name>[^,]+) currently residing at"],
            "type": "string"
        },
        "tenant_current_address": {
            "patterns": ["currently residing at Address (?P<address>[^.]+)"],
            "type": "string"
        },
        "monthly_rent_amount": {
            "patterns": ["rent of RS.(?P<amount>[\\d,]+) per month", "monthly rent of
RS.(?P<amount>[\\d,]+)"],
            "type": "int"
        },
        "lease_duration": {
            "patterns": ["for a period of (?P<months>\\d+) months only"],
            "type": "int"
        },
        "lease_start_date": {
            "patterns": ["commencing from (?P<date>\\d{2}/\\d{2}/\\d{4})"],
            "type": "date"
        },
        "rental_advance_amount": {
            "patterns": ["advance rent Rs.(?P<amount>[\\d,]+)"],
            "type": "int"
        },
        "property_address": {
            "patterns": ["Address: (?P<address>[^,]+), \\(residential address of the landlord\\)"],
            "type": "string"
        },
        "rental_address": {
            "patterns": ["currently residing at Address (?P<address>[^.]+)"],
            "type": "string"
        },
        "lease_period": {
            "patterns": ["for a period of (?P<months>\\d+) months"],
            "type": "int"
        }
    }

    extractor = EntityExtractor(rules)
    entities = extractor.extract(text)
    for key, value in entities.items():
        details[key] = value

    return details

rental_agreement_text = """
RENTAL AGREEMENT
This Rent Agreement is made on this 15/09/2019 (date of rent agreement) by Latha K
Rao (name of the landlord) D/O Krishnaswamy (father's name of the landlord),
Address: Flat-G2, Guru Saravana Apartment, Andavarnagar, 1st street, Kodambakkam,
Chennai. Herein after called the Lessor / Owner,
Party Of the first part & with Lessee/Tenant V. Praveen, S/o S Vivekanandhan currently
residing at Address 50/109, 2nd floor, Dr Natesan Road, Triplicane, Chennai 600005.
That the expression of the term , Lessor/Owner and the Lessee/Tenant Shall mean
and include their legal heirs successors , assigns , representative etc. Whereas the
Lessor /Owner is the owner and in possession of the flat and has agreed to let out
the Two Bed rooms, Hall and Kitchen (2BHK) flat to the Lessee/Tenant and the Lessee/Tenant
has agreed to take the same on rent of Rs.15,000/- (Rupees Fifteen thousand only) per month.
NOW THIS RENT AGREEMENT WITNESSETH AS UNDER:-
1. That the Tenant/Lessee shall pay as the monthly rent of RS.15,000/- per month
on 1st day of every month, excluding electricity and maintenance charge.
... (additional agreement details)
"""

rental_details = extract_rental_agreement_details(rental_agreement_text)

print("Extracted Rental Agreement Details:")
print("----------------------------------")
for key, value in rental_details.items():
    print(f"{key.replace('_', ' ').title()}: {value}")
```



```python
import re
def extract_rental_agreement_details(text):
    details = {}
    landlord_pattern = r"by\s+(.*?)\s+\(name of the landlord\)\s+(.*?)\s+\
    landlord_match = re.search(landlord_pattern, text, re.DOTALL)
    if landlord_match:
        details['landlord_name'] = landlord_match.group(1).strip()
        details['landlord_father_name'] = landlord_match.group(2).strip()
        details['landlord_address'] = landlord_match.group(3).strip()
    tenant_pattern = r"with\s+Lessee/Tenant\s+(.*?),\s+S/o\s+(.*?)\s+currer
    tenant_match = re.search(tenant_pattern, text, re.DOTALL)
    if tenant_match:
        details['tenant_name'] = tenant_match.group(1).strip()
        details['tenant_father_name'] = tenant_match.group(2).strip()
        details['tenant_current_address'] = tenant_match.group(3).strip()
    rent_pattern = r"monthly\s+rent\s+of\s+RS\.(.*?)\s+per\s+month"
    rent_match = re.search(rent_pattern, text, re.IGNORECASE)
    if rent_match:
        details['rent_amount'] = rent_match.group(1).strip()
    duration_pattern = r"for\s+a\s+period\s+of\s+(\d+)\s+months\s+only"
    duration_match = re.search(duration_pattern, text, re.IGNORECASE)
    if duration_match:
        details['lease_duration'] = duration_match.group(1).strip()
    start_date_pattern = r"commencing\s+from\s+(\d+\s+\w+\s+\d{4})"
    start_date_match = re.search(start_date_pattern, text, re.IGNORECASE)
    if start_date_match:
        details['lease_start_date'] = start_date_match.group(1).strip()
    advance_rent_pattern = r"advance\s+rent\s+Rs\.(.*?)\s+\(Rupees"
    advance_rent_match = re.search(advance_rent_pattern, text, re.IGNORECA
    if advance_rent_match:
        details['advance_rent_amount'] = advance_rent_match.group(1).strip
    return details
rental_agreement_text = """
RENTAL AGREEMENT
This Rent Agreement is made on this 15/09/2019 (date of rent agreement) by
Rao (name of the landlord) D/O Krishnaswamy (father's name of the landlord
Address: Flat-G2, Guru Saravana Apartment, Andavarnagar, 1st street, Kodam
Chennai (residential address of the landlord). Herein after called the Les
Party Of the first part & with Lessee/Tenant V. Praveen, S/o S Vivekanandha
residing at Address 50/109, 2nd floor, Dr Natesan Road, Triplicane, Chenna
That the expression of the term , Lessor/Owner and the Lessee/Tenant Shall
```

```
Python 3.12.4 (tags/v3.12.4:8e8a4ba, Jun  6 2024, 19:30:16) [MSC v.1940 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: C:/Users/TEM240000376/Downloads/rentals.py
Extracted Rental Agreement Details:
----------------------------------
Landlord Name: Latha K
Rao
Landlord Father Name: D/O Krishnaswamy
Landlord Address: Flat-G2, Guru Saravana Apartment, Andavarnagar, 1st street, Ko
dambakkam,
Chennai
Tenant Name: V. Praveen
Tenant Father Name: S Vivekanandhan
Tenant Current Address: 50/109,
Rent Amount: 15,000/-
Lease Start Date: 15 Sept 2021
Advance Rent Amount: 75,000/-
>>>
```

5. Best Practices Learned

Best Practices in Selenium Automation Script and Text Extraction

Selenium Automation - Best Practices

In the realm of Selenium automation, adhering to best practices is pivotal for crafting scripts that are both robust and reliable. One of the primary best practices involves effective error handling. Web automation scripts often encounter unexpected issues such as elements not being found or timeouts occurring due to slow page loads. Implementing try-except blocks helps manage these potential errors gracefully. This method ensures that the script can handle exceptions, such as NoSuchElementException or TimeoutException, without crashing. By catching these exceptions and managing them appropriately—such as logging the issue or retrying the operation—the script maintains its stability and continues functioning smoothly.

Another crucial best practice is the use of explicit waits. In dynamic web environments, elements may not be immediately available for interaction due to various reasons, including slow network conditions or rendering delays. Explicit waits address this by allowing the script to pause until specific conditions are met, such as the presence or visibility of an element. Utilizing Selenium's WebDriverWait in conjunction with expected conditions, such as EC.presence_of_element_located or EC.visibility_of_element_located, ensures that interactions occur only when elements are fully prepared. This approach helps prevent errors related to attempting actions on elements that are not yet available and improves the reliability of the automation process.

Comprehensive logging is another best practice that plays a significant role in maintaining and debugging Selenium scripts. Effective logging captures detailed information about each step of the script's execution, including actions taken and any encountered errors.

By keeping track of these details, logs provide insights into the script's behavior and facilitate the identification and resolution of issues. Detailed logs can significantly aid in troubleshooting by providing a clear record of what happened during the script’s execution.

Implementing retry mechanisms is also beneficial for handling transient issues. Automation scripts often interact with web elements or external systems that might occasionally fail due to temporary issues, such as network glitches or server delays. By incorporating retry logic, the script can automatically attempt to perform a failed operation multiple times before concluding a failure. This approach enhances the script's robustness and reduces the likelihood of failure due to temporary disruptions.

Text Extraction Tool - Best Practices

When developing a text extraction tool, especially one that employs regular expressions (regex) and structured rules, several best practices are essential for ensuring accuracy and efficiency. One key practice is the use of well-defined regular expressions. Regular expressions are powerful tools for pattern matching, allowing the extraction of specific data from complex text documents. By crafting precise and clear patterns for different types of data—such as dates, names, or addresses—the tool can effectively identify and extract relevant information while minimizing errors and inaccuracies.

Another best practice is modularizing the code into functions and classes. This practice enhances code readability and maintainability by organizing the extraction logic into discrete components. Each function or class should be responsible for a specific task or aspect of the extraction process, making the code easier to manage and update. Modular code also promotes reusability, as functions or classes can be reused across different parts of the project or in future projects.

Testing and validating regular expressions and extraction rules are crucial for ensuring their correctness. Before deploying the text extraction tool, it is essential to test the regular expressions with a variety of sample documents to verify their effectiveness. This testing helps to identify any issues with the patterns and allows for adjustments to improve accuracy. Regular validation ensures that the extraction tool performs reliably across different types of input.

Lastly, maintaining thorough documentation is vital for the long-term success of the text extraction tool. Documentation should detail the purpose and functionality of each regular expression, the structure of the extraction rules, and any assumptions or limitations. Well-documented code helps future developers understand the logic behind the extraction process and facilitates easier modifications or enhancements.

By following these best practices, the development of Selenium automation scripts and text extraction tools can achieve higher reliability, accuracy, and maintainability. Implementing these practices ensures that the scripts and tools operate efficiently and effectively, meeting the needs of users and stakeholders.

6, Conclusion

The project successfully achieved its objectives by creating robust tools for web automation and text extraction. The Selenium automation script demonstrated how to interact programmatically with a web page to retrieve and process information, while the entity extraction tool showcased the ability to parse and structure data from complex text documents. The implemented solutions are not only functional but also adhere to best practices in coding and automation, ensuring reliability and efficiency.

Both components of the project illustrate the power of Python and its libraries in automating repetitive tasks and extracting valuable information from unstructured data. These solutions can be adapted and extended for various other applications, showcasing the versatility of modern programming techniques.

References

Selenium Documentation: https://www.selenium.dev/documentation/

Python re Module Documentation:
https://docs.python.org/3/library/re.html

webdriver_manager Package Documentation:
https://pypi.org/project/webdriver-manager/

Regular Expressions Cookbook by Jan Goyvaerts and Steven Levithan