UNIT - III

📝 **Topic: Advanced Web Scraping, Reading Documents & Document Encoding**

---

1. ADVANCED SCRAPING

☑ **Definition:**

Web scraping is a method to extract data from websites using automated scripts or tools, especially when data is spread across multiple web pages.

☑ **Key Points:**

- **Dynamic Sites:** Tools like Selenium, Playwright, and Puppeteer simulate user actions (clicks, scrolls, form submissions).
- **AJAX & API Handling:** AJAX/Background APIs are used for loading data dynamically without reloading the page.
- **Session Handling:** Use requests.Session() for login, cookies, and user-authenticated content.
- **Anti-Bot Protection:** Use techniques to handle CAPTCHA, IP blocking, and rotate user agents/proxies to avoid detection.
- **Scrapy Framework:** Used for large-scale scraping. Handles retries, proxies, headers, etc.
- **Data Formats:** Output can be saved in CSV, JSON, SQLite, MongoDB.

📌 **Example Tools/Techniques:**

- requests, BeautifulSoup, Scrapy, Selenium, Playwright

---

2. READING DOCUMENTS

☑ **Definition:**

Reading documents in web scraping refers to extracting data from non-HTML content like PDFs, DOCs, CSVs, etc.

☑ **Key Points:**

- The internet isn't only HTML—includes files like PDFs, Word docs, images, videos, etc.
- Web scraping is not only about reading text but collecting structured and unstructured data.
- Libraries used:
  - **PDFs:** PyPDF2, pdfplumber
  - **Word docs:** python-docx
  - **CSV/Excel:** csv module, pandas

📌 **Important Uses:**

- Extract data like names, tags, prices, or IDs from various formats.

- Clean and filter irrelevant content using regex or parsing tools.

---

3. DOCUMENT ENCODING

✅ **Definition:**

Document encoding ensures the text extracted from a website is displayed correctly and prevents errors like garbled or unreadable text.

✅ **Key Points:**

- Text from websites is in binary format → needs decoding into readable text.
- Common encodings: **UTF-8**, **ISO-8859-1**, **Windows-1252**
- Encoding errors lead to issues like seeing "AØ" instead of "é".
- Proper encoding is essential for **accuracy and usability** of scraped data.

✅ **10-Mark Answer Structure (Suggestion):**

Introduction:

- Define web scraping.
- Mention that modern scraping involves automation, session handling, reading files, and encoding.

Body:

- Explain dynamic scraping tools (Selenium, Scrapy).
- Highlight session handling and anti-bot measures.
- Discuss reading document types (PDFs, CSVs).
- Explain the importance of encoding.

Conclusion:

- Summarize how advanced scraping is crucial for gathering accurate, diverse data across the internet.

✅ **Topic: Document Formats and Parsing in Python**

📌 **1. Common Document Encodings**

Definition:

Document encodings define how characters are represented in bytes when storing or transmitting data.

UNIT - III

Key Points:

- **ASCII**: Old encoding supporting 128 characters (7-bit).
- **ISO-8859-1 (Latin-1)**: 8-bit; supports Western European characters.
- **Windows-1252**: Similar to Latin-1; common in Windows environments.
- **UTF-8**: Most widely used; supports characters from every writing system.
    - It's **variable-width** and **backward compatible** with ASCII.

📌 2. Text Files

Definition:

Plain text files (.txt) contain unformatted, human-readable characters and are commonly used for data storage.

Code Example:

```
from urllib.request import urlopen

textPage = urlopen(url)

print(textPage.read())
```

Important Notes:

- HTML can be converted into a BeautifulSoup object for parsing.
- Parsing is harder if HTML is missing or malformed.
- Extracting specific data requires careful string analysis.

📌 3. CSV Files (Comma-Separated Values)

Definition:

CSV is a **plain text format** where data is stored in **tabular form** using commas (,) or semicolons (;).

Key Features:

- Easy to read and write in Python.
- No formatting or formulas—just raw data.

Code Example:

```
from urllib.request import urlopen

from io import StringIO

import csv
```

```
data = urlopen('myfile.csv').read().decode('ascii', 'ignore')

dataFile = StringIO(data)

csvReader = csv.reader(dataFile)


for row in csvReader:

   print(row)
```

Output:

```
['Name', 'Year']

["Monty Python's Flying Circus", '1970']

["Another Monty Python Record", '1971']

["Monty Python's Previous Record", '1972']
```

📌 **4. PDF Files**

Definition:

PDF (Portable Document Format) was introduced by Adobe in 1993 to enable platform-independent document viewing.

Key Points:

- Supports both **text and image content**.
- Requires specialized libraries for parsing (e.g., PyPDF2, pdfplumber, PyMuPDF).

Use in Python:

Used to extract:

- Text
- Metadata
- Structure (reports, invoices, forms)

📝 **10-Marks Exam Highlights (Bullet Points)**

- **Document encoding** determines how characters are stored in files.
- Common encodings: ASCII, UTF-8, ISO-8859-1, Windows-1252.
- **Text files** (.txt) are basic readable formats; often used in web scraping.
- Use urllib and BeautifulSoup to process HTML from web pages.
- **CSV** files store tabular data using commas or semicolons.

- Python's csv module allows reading and writing CSV files easily.
- StringIO is used to treat strings like files in memory.
- **PDFs** are structured documents requiring tools like PyPDF2, pdfplumber, or fitz.
- Parsing PDFs helps extract data from reports, invoices, etc.
- Understanding file formats is essential for **data extraction, scraping, and analysis** in Python.

☑ Topic: Working with PDF and Word Documents in Python

📌 1. PDF Parsing in Python

📖 Definition:

PDF parsing is the process of extracting text, metadata, and tables from PDF files using Python libraries.

🔧 PDF Parsing Libraries and Usage

1. Using PyPDF2

- Basic library to extract text and metadata.
- Does not support tables or images.

Code Snippet:

```
from PyPDF2 import PdfReader
with open('sample2.pdf', 'rb') as file:
  reader = PdfReader(file)
  print(len(reader.pages))
  text = reader.pages[0].extract_text()
  print(text)
```

---

2. Using pdfplumber ***(Recommended for structured text/tables)***

- Can extract **text and tables** accurately.
- Very useful for reading invoices, structured PDFs.

Code Snippet:

```
import pdfplumber
with pdfplumber.open('sample.pdf') as pdf:
  page = pdf.pages[0]
  text = page.extract_text()
  print(text)
  table = page.extract_table()
  print(table)
```

---

3. Using PyMuPDF (fitz)

- Advanced PDF processing.
- Supports images, text extraction, and more.

Code Snippet:

```
import fitz  # PyMuPDF
doc = fitz.open("sample.pdf")
page = doc[0]
text = page.get_text()
print(text)
```

📌 **2. Microsoft Word and .DOCX Files**

📖 **Definition:**

.docx is the file format used by Microsoft Word for documents containing formatted text, tables, and media.

📌 **Key Library:**

- **python-docx** is used to read .docx files programmatically.

🔧 **Important Functionalities:**

a) Reading Document Structure

```
from docx import Document
doc = Document("example.docx")
```

b) Reading Paragraphs, Headings, Styles

```
for para in doc.paragraphs:
  if para.style.name.startswith('Heading'):
    print("Heading:", para.text)
    print("Paragraph:", para.text)
```

c) Reading Tables

```
for table in doc.tables:
    for row in table.rows:
        for cell in row.cells:
            print(cell.text, end='\t')
        print()
```

d) Reading Bold/Italic Text

```
for para in doc.paragraphs:
    for run in para.runs:
        if run.bold:
            print("Bold:", run.text)
        elif run.italic:
            print("Italic:", run.text)
```

📝 **10-Marks Exam Answer: Highlight Points**

- Python offers libraries to extract content from **PDF** and **Word** documents.
- **PyPDF2**: Used for basic PDF text extraction.
- **pdfplumber**: Extracts text + structured tables (best for formatted PDFs).
- **PyMuPDF (fitz)**: Extracts text/images; faster and more advanced.
- .docx files (Word documents) are parsed using **python-docx**.
- python-docx allows reading:
    - Paragraphs
    - Headings
    - Tables
    - Bold/Italic formatting
- Tables are read using nested loops over doc.tables → table.rows → row.cells.
- This is useful for scraping legal documents, reports, and structured datasets.

UNIT - III

☑ **Topic: Natural Language Processing (NLP), Summarizing Data, and Markov Models**

📌 1. Reading and Writing Natural Languages

📖 Definition:

Natural Language Processing (NLP) is a branch of AI that deals with the interaction between computers and human (natural) languages like English, Hindi, etc.

🔍 Key Applications:

- Text understanding
- Text translation
- Text summarization
- Sentiment analysis
- Chatbots, voice assistants, etc.

🛠 Key Concepts:

- **Fetching** web pages and extracting natural language data.
- **NLTK (Natural Language Toolkit)**: A popular Python library used for NLP tasks.
- **Contextual Analysis**: Identifying relationships between words.
- **Predictive Analytics**: Using text for document classification, intent prediction, etc.
- **Evaluation Metrics**: Used to measure NLP model quality.

📌 2. Summarizing Data

📖 Definition:

Summarizing data involves identifying the **most frequently used word sequences** (n-grams) to extract the essence of text.

🔍 Purpose:

- Generate short summaries.
- Extract key points from longer documents.

🔢 Example Method:

- Use **n-grams** (commonly bi-grams or 2-grams).
- Locate **popular phrases** in the text to summarize content.

Example Output (most frequent 2-grams from a U.S. Constitution extract):

- "The Constitution"

- “The executive”
- “The General”
- “Having emerged”
- “Of my life”

📌 **3. Markov Models**

📖 **Definition:**

A **Markov Model** is a probabilistic model used to predict the next event (state) based only on the current state.

📉 **Real-world Example:**

Used in:

- Text generation
- Weather forecasting
- Spam filtering
- AI decision systems

Example:

- Weather transitions: Sunny (70%) → Sunny (25%), Cloudy (20%), Rainy (10%)
- Markov chain represented using **state diagram** (as seen in the image).

💥 **Characteristics of Markov Model:**

- Each state transition is based **only on the current state** (memoryless).
- All outgoing probabilities from a state must **sum to 100%**.
- Can simulate complex, random, but structured systems.
- Can be used to **generate infinite sequences** like weather or text.

📝 **10-Marks Exam Highlights (Bullet Format)**

- **NLP** helps machines understand, interpret, and generate human languages.
- **NLTK** is the most widely used Python toolkit for NLP tasks.
- Text summarization extracts key points using **frequent word sequences (n-grams)**.
- Summarizing helps in **content analysis**, **document indexing**, and **search**.
- A **Markov Model** uses **state-to-state transition probabilities**.
- Markov models are **memoryless** – next state depends only on the current one.

- Applications: **Weather prediction, text generation, game AI**.
- Markov diagrams show **nodes as states** and **arrows as transitions with probabilities**.
- Each node's outgoing probabilities must **sum to 100%**.
- Markov chains are useful for analyzing and simulating **sequential systems**.

Topic: Natural Language Toolkit (NLTK)

Definition:
NLTK (Natural Language Toolkit) is a popular Python library used for processing human language data. It supports many NLP tasks like tokenization, stop word removal, stemming, lemmatization, POS tagging, and named entity recognition.

Important Points and Highlights for Exam:

1. **NLTK Overview:**
   - NLTK is widely used for natural language processing (NLP).
   - It supports breaking text into words or sentences (tokenization).
   - It helps remove common words (stopwords) which do not add meaning.
   - Stemming and Lemmatization reduce words to their root forms.
   - Part-of-Speech (POS) tagging labels words as nouns, verbs, etc.
   - Named Entity Recognition (NER) identifies proper names like people and places.
   - It contains built-in corpora like the Gutenberg corpus for practice.
   - Useful for beginners and researchers alike.
2. **NLTK Installation:**
   - Installed via Python package managers.
   - import nltk
   - Download required datasets using nltk.download()
3. **Key NLTK Features with Examples:**
   - **Tokenization:**
   - from nltk.tokenize import word_tokenize, sent_tokenize
   - text = "Natural Language Processing is fun."
   - print(word_tokenize(text)) # Splits into words
   - print(sent_tokenize(text)) # Splits into sentences
   - **Stopword Removal:**

- from nltk.corpus import stopwords
- stop_words = set(stopwords.words('english'))
- words = word_tokenize("This is a sample sentence.")
- filtered = [w for w in words if w.lower() not in stop_words]
- print(filtered) # Words excluding stopwords
- **Stemming and Lemmatization:**
- from nltk.stem import PorterStemmer, WordNetLemmatizer
- stemmer = PorterStemmer()
- print(stemmer.stem("running")) # Outputs 'run'
- lemmatizer = WordNetLemmatizer()
- print(lemmatizer.lemmatize("running", pos='v')) # Outputs 'run'
- **POS Tagging:**
- from nltk import pos_tag
- print(pos_tag(word_tokenize("NLTK is a leading toolkit for NLP.")))
- **Named Entity Recognition (NER):**
- from nltk import ne_chunk
- from nltk.tree import Tree
- sentence = "Barack Obama was born in Hawaii."

4. **NLTK Text Object:**
    - Text object lets you do advanced analysis on texts.
    - Created using:
    - from nltk import Text

Topic: Web Scraping — Crawling through Forms and Logins

Definition:
Web scraping involves extracting data from websites. Advanced scraping includes submitting forms and handling user logins, which are often required for accessing protected content.

Important Points:

- Crawling forms and logins is essential for scraping content behind authentication.
- Uses libraries like requests for submitting form data programmatically.
- To handle login sessions, maintain cookies and headers with Session objects.

- Must deal with CSRF tokens and hidden fields in forms.
- JavaScript rendering might require tools like Selenium for full browser automation.
- Effective crawling includes form data inspection, session handling, and sometimes browser automation.

Example of HTML form to be handled in scraping:

```
<form method="post" action="processing.php">

First name: <input type="text" name="firstname"><br>

Last name: <input type="text" name="lastname"><br>

<input type="submit" value="Submit">

</form>
```

Summary for Exam Answer:

- Define NLTK and its importance in NLP.
- Mention key NLP tasks supported by NLTK (tokenization, stopwords, stemming, POS tagging, NER).
- Provide small example code snippets for important features.
- Define web scraping and explain the challenges in scraping forms and logins.
- Highlight techniques for handling session, CSRF tokens, and JavaScript-driven content.
- End with a note on the importance of understanding HTML forms for scraping.

Topic: Python Requests Library

Definition:

- The Requests library in Python is a powerful and user-friendly HTTP client used to send HTTP/1.1 requests such as GET, POST, PUT, DELETE, etc.
- It is commonly used for web scraping, REST API interactions, and automation tasks that require communication with web servers.

Important Points:

1. **Purpose and Usage:**
    - Helps in fetching web pages, submitting forms, handling cookies, and working with JSON data.
    - Simplifies sending HTTP requests compared to Python's built-in urllib.
2. **Common HTTP Methods:**

- **GET**: Used to retrieve data from a server.
- **POST**: Used to submit form data or login information.
- Others include PUT, DELETE, etc.

3. **Example Usage:**
    - Sending a GET request:
    - import requests
    - response = requests.get('https://example.com')
    - print(response.status_code) # e.g., 200 OK
    - print(response.text)   # HTML content of the response
    - Sending a POST request:
    - data = {'username': 'user1', 'password': 'pass123'}
    - response = requests.post('https://example.com/login', data=data)
    - print(response.text)
4. **Status Codes:**
    - 200: OK (Successful)
    - 301: Moved Permanently
    - 302: Found (Redirect)
    - 400: Bad Request
    - 401: Unauthorized
    - 403: Forbidden
    - 404: Not Found
    - 500: Internal Server Error
5. **URL Parameters and Query Strings:**
    - Example:
    - params = {'q': 'python', 'sort': 'recent'}
    - response = requests.get('https://example.com/search', params=params)
    - print(response.url) # URL with query strings appended
6. **Custom Headers:**
    - Headers like User-Agent are often required when scraping:
    - headers = {'User-Agent': 'Mozilla/5.0'}
    - response = requests.get('https://example.com', headers=headers)

7. **Sending JSON Data:**
   - Use json parameter instead of data for sending JSON:
   - json_data = {'id': 101, 'name': 'John'}
   - response = requests.post('https://api.example.com/add', json=json_data)

Summary for 10 Marks:

- **What is Python Requests Library?**
  A simple, powerful Python library for making HTTP requests to communicate with web servers.
- **Main HTTP methods supported:**
  GET (retrieve data), POST (submit data), PUT, DELETE, etc.
- **Features:**
  Easy to use, supports URL parameters, headers, JSON data, cookies, sessions, and error handling.
- **Example of GET and POST requests with syntax.**
- **Important HTTP status codes and their meanings.**
- **Usage of query parameters, custom headers, and sending JSON data in requests.**

Topic: Web Scraping Using Python's Requests Library

---

1. Definition:

Web scraping involves programmatically extracting data from websites. Python's requests library helps to send HTTP requests like GET or POST to interact with web pages, handle sessions, cookies, and form submissions.

---

2. Key Attributes of requests Response:

Attribute	**Description**
response.text	Response body as text
response.content	Response body as bytes
response.status_code	HTTP status code (e.g., 200, 404)
response.headers	Metadata about the response
response.json()	Convert JSON response to Python dictionary

UNIT - III

3. Working with Cookies and Sessions:

- Cookies can be viewed using print(response.cookies).
- To maintain a login session, use session = requests.Session().
- Example to login:
- session.post('https://example.com/login', data={'user': 'admin', 'pass': '1234'})
- profile = session.get('https://example.com/profile')

---

4. Timeout and Exception Handling:

- Timeout example:
- response = requests.get('https://example.com', timeout=5)
- Handle errors using try-except:
- try:
-     response.raise_for_status() # Raises error for bad responses (4xx, 5xx)
- except requests.exceptions.RequestException as e:
-     print("Error:", e)

---

5. File Upload and Download:

- **Upload a file:**
- files = {'file': open('report.pdf', 'rb')}
- response = requests.post('https://example.com/upload', files=files)
- **Download a file:**
- response = requests.get('https://example.com/file.pdf')
- with open('file.pdf', 'wb') as f:
-     f.write(response.content)

---

6. Advantages of Python's requests library:

- Easy syntax and user-friendly.
- Supports sessions and cookies.
- Allows customization with headers and parameters.
- Handles JSON and file uploads easily.
- Provides robust error-handling features.

7. Submitting a Basic Form:

- A basic form submission mimics a user filling and submitting an HTML form.
- You can use either GET or POST method to send form data.

Example HTML form:

```
<form method="POST" action="/login">
  <input type="text" name="username">
  <input type="password" name="password">
  <input type="submit" value="Login">
</form>
```

Python code to submit form data using POST:

```
import requests

url = 'https://example.com/login'

payload = {
    'username': 'myuser',
    'password': 'mypassword'
}

response = requests.post(url, data=payload)
print(response.text)  # Display success message or HTML content
```

8. Note on Headers:

- Websites may block automated scripts unless proper headers like User-Agent are sent.
- Example:
- headers = {
-     'User-Agent': 'Mozilla/5.0'
- }

UNIT - III

Topic: Working with Web Forms Using Python

---

1. Form Submission and Requests

- Web forms send data through URLs or POST requests.
- Example GET request URL:
- https://example.com/search?q=python&category=books
- Python's requests library can send and handle these requests:
- response = requests.get(url, params=params)
- print(response.url)
- Sessions in Python (requests.session()) help maintain login across pages by storing cookies.

---

2. Login Form Example

- URL and login data can be set in a dictionary:
- login_url = "https://example.com/login"
- login_data = {'username': 'user1', 'password': 'pass123'}
- Use session.post() to send login data and maintain session:
- session.post(login_url, data=login_data)

---

3. Summary of Required Elements for Form Submission

Element	**Purpose**
url	Endpoint where form is submitted
data	Data containing input field values
headers	Optional headers (User Agent, Referer)
session	Maintain cookies & login session

---

4. Radio Buttons

- Allow users to select one option from multiple choices.
- All radio buttons in a group share the same name but different values.
- Example:
- <input type="radio" name="gender" value="male">

- <input type="radio" name="gender" value="female">
- To submit selected value using Python:
- data = {'gender': 'female'}
- requests.post(url, data=data)

---

5. Checkboxes

- Allow multiple options to be selected independently.
- Checkboxes may share the same or different names.
- Example:
- <input type="checkbox" name="hobby" value="reading">
- <input type="checkbox" name="hobby" value="sports">
- Submit values as list:
- data = {'hobby': ['reading', 'sports']}
- requests.post(url, data=data)

---

6. Other Input Types

Input Type	**Description**	**Python Handling**
text	Single-line input field	{'name': 'value'}
password	Secure password input	{'password': 'mypassword'}
hidden	Hidden input (e.g. CSRF tokens)	Must be included in submission
select (dropdown)	Selecting from list	{'country': 'India'}
file	Upload file	Use files argument in requests
textarea	Multi-line text field	{'comments': 'Great job'}

---

7. Finding Values Using BeautifulSoup

- Use BeautifulSoup to parse HTML and extract selected radio/checkbox value:
- from bs4 import BeautifulSoup
-
- html = requests.get(url).text
- soup = BeautifulSoup(html, 'html.parser')

- selected = soup.find('input', {'type': 'radio', 'checked': True})['value']

---

8. Submitting Files and Images

- Uploading files/images via web forms using Python is common.
- This is used for web automation and scraping when interacting with upload features.

Summary for Exam (10 Marks):

- Understand how forms submit data using GET or POST methods.
- Use Python requests to send form data and maintain session cookies.
- Know HTML input types: radio buttons (single choice), checkboxes (multiple choices), text, password, hidden, file upload.
- Handle input types correctly when automating form submissions.
- Use BeautifulSoup to extract data from web forms.
- File/image upload automation is important in web scraping.

Web Scraping

Definition:
Web scraping is the process of extracting data from websites. It often involves handling sessions, cookies, and form submissions to access protected data.

Handling Logins and Cookies

Definition:
Managing logins and cookies is essential to access protected content on websites by maintaining session states and authentication details.

Key Points:

- Use the same session after login to access protected pages.
- Cookies store session information automatically.
- The Python requests library with requests.Session() is commonly used.
- Sessions maintain cookies and headers automatically.
- Handle CSRF tokens by scraping them from login pages and including them in POST requests.

Basic Login with Session Handling (Python example)

```
import requests

session = requests.Session()

login_url = 'https://example.com/login'

login_data = {'username': 'user123', 'password': 'mypassword'}

response = session.post(login_url, data=login_data)

print(response.status_code)  # Check if login was successful


dashboard = session.get('https://example.com/dashboard')

print(dashboard.text)  # Access protected content
```

File Upload

Definition:
Uploading files through forms (e.g., profile pictures, resumes) is done using a multipart/form-data POST request.

Key Points:

- Use <input type="file" name="file"> in HTML forms.
- In Python requests, use the files parameter to send files.
- Example:

```
import requests

url = 'https://example.com/upload'

file = {'file': open('example.jpg', 'rb')}

response = requests.post(url, files=file)

print(response.text)
```

Error Handling and Response Checks

Key Points:

- Use try-except blocks to catch upload errors.
- Check response status with response.raise_for_status() to confirm success.

```
try:

   response = requests.post(url, files=files, timeout=30)

   response.raise_for_status()

   print("File uploaded successfully")
```

```
except requests.exceptions.RequestException as e:
    print("Upload failed:", e)
```

Real-World Use Cases

Use Case	**Example**
Resume submission	Job portals
Profile picture	Social networks
Bulk upload	E-commerce galleries
Form attachment	Helpdesk forms, feedback forms
