**DAMG 7245 - Assignment 2 - Team 5**

| **Summary** | This codelab encompasses web scraping from the CFA Institute’s website to gather specific content, extracting text from PDFs, uploading structured data and extracted text to a Snowflake database, and integrating with AWS S3 for storage |
| --- | --- |
| **URL** | https://codelabs-preview.appspot.com/?file\_id=1H02nPi64xmAH9DNvVs2ri15O0f36IX3hWvasRI22rFc#0 |
| **Category** | Databases |
| **Environment** | Python |
| **Status** | Completed |
| **Github** | <https://github.com/BigDataIA-Spring2024-Sec1-Team5/Assignment2> |
| **Authors** | Aditya Kanala, Shikhar Patel, Shubh Patel |

**Table of Contents**

[Architecture and Workflow](#_vjfa17joz2h8)

[Overview](#_70p7luabbmg3)

[Workflow](#_hqgc53ofwerd)

[Web Scraping and Dataset Creation](#_lre49l2hb00x)

[PDF Extraction](#_20y1pq7zuj6d)

[PyPDF](#_vt22sd9b3bl7)

[Grobid](#_rj2sut67dsim)

Metadata Process

Database Upload

Cloud Storage Integration

References

**Architecture**

# **Architecture and Workflow**

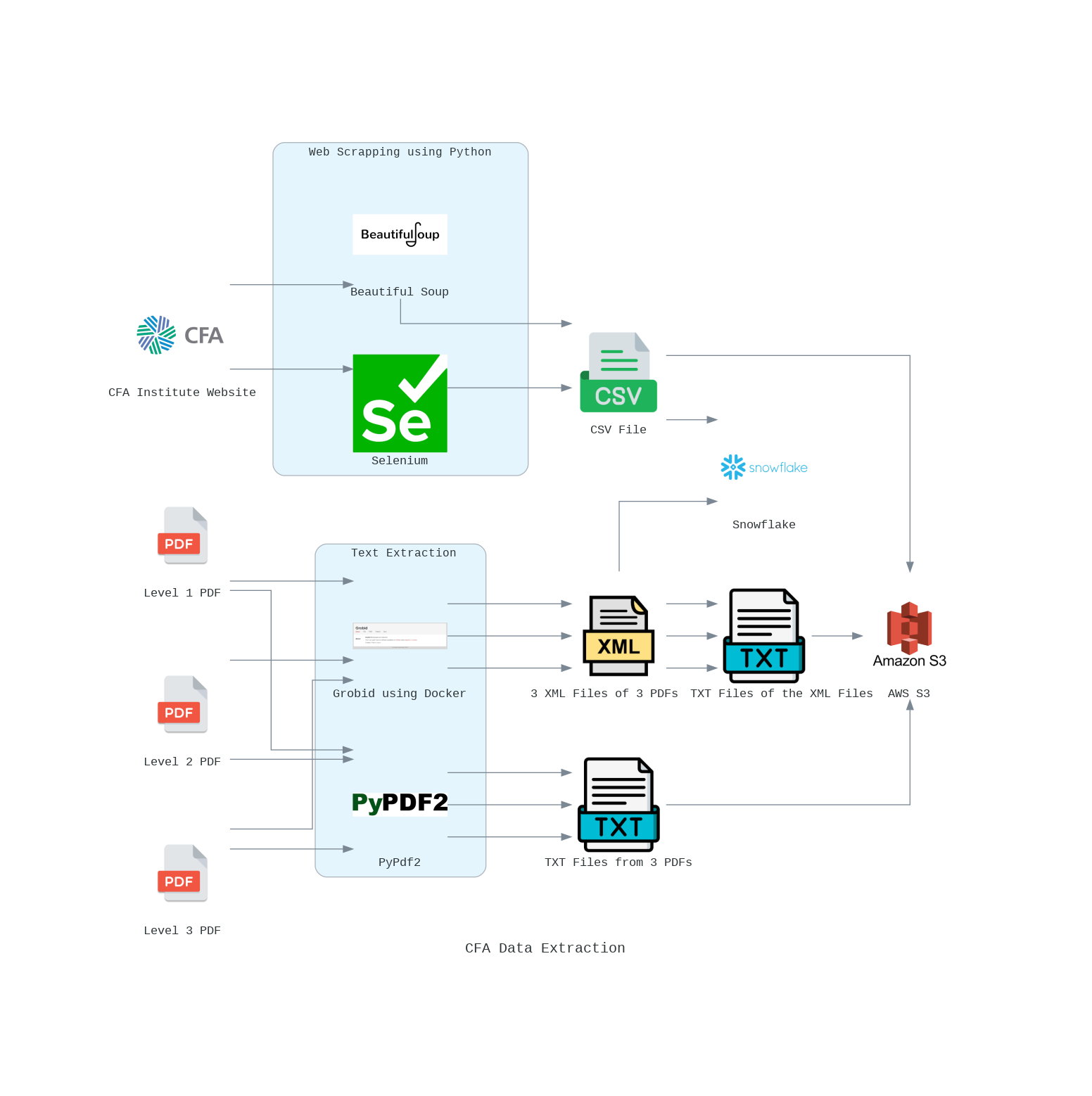
#### **Overview**:

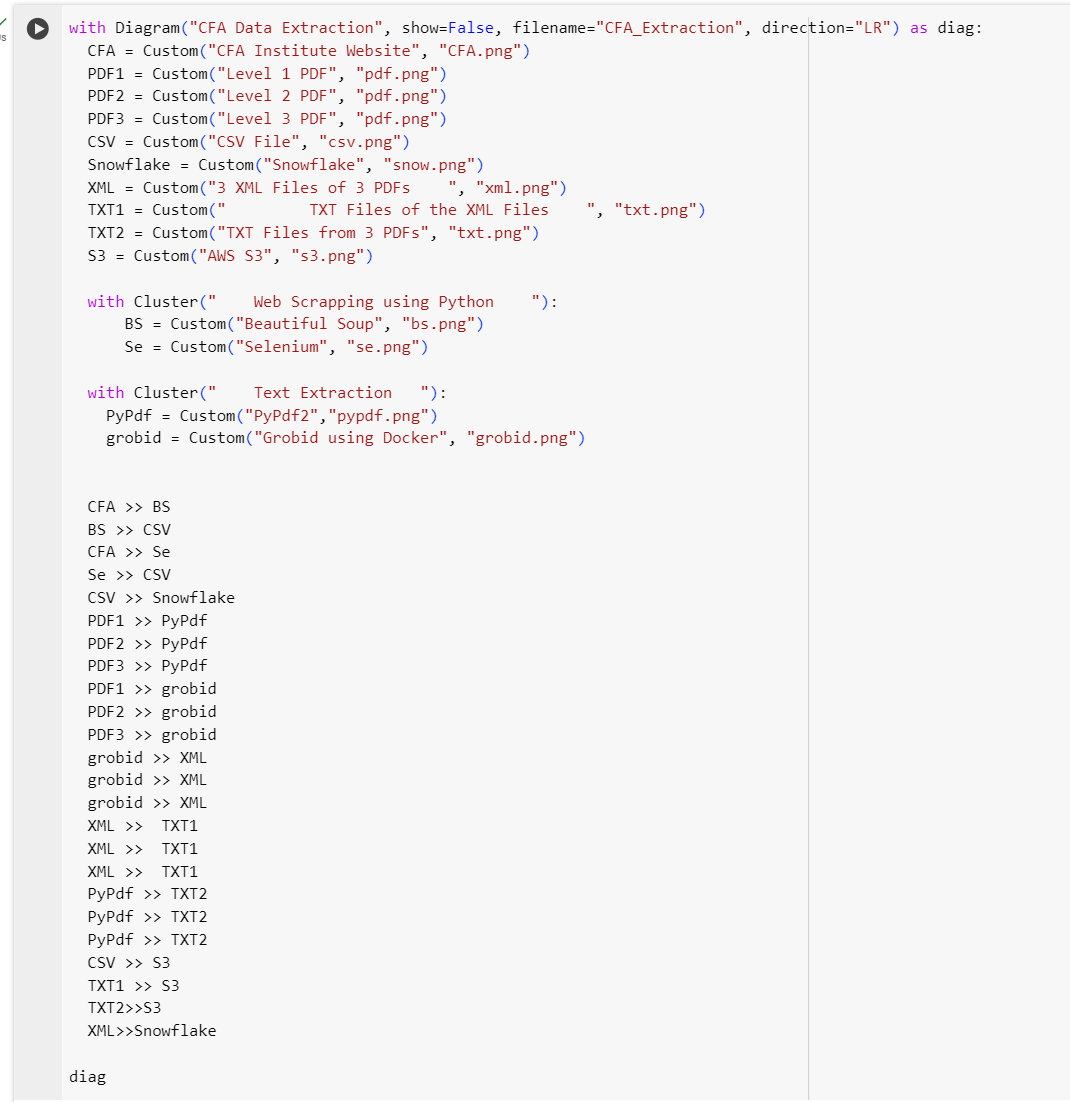
The CFA Data Extraction process is designed to automate the collection and processing of educational materials from the CFA Institute Website. This process involves web scraping to gather data, extracting text from PDF documents, and storing the processed data in various formats for further analysis and storage. The workflow is visualized using a diagram that outlines the key components and their interactions.

#### **Components:**

1. **CFA Institute Website**: The primary source of reading materials, including PDF documents for different levels of CFA examinations.
2. **Beautiful Soup & Selenium**: Libraries used for web scraping. Beautiful Soup parses HTML and XML documents, while Selenium automates web browser interaction to access and retrieve data from the CFA Institute Website.
3. **PyPdf2 & Grobid using Docker**: Tools for extracting text from PDF documents. PyPdf2 is a Pure-Python library built as a PDF toolkit. Grobid is an advanced tool that uses machine learning to convert PDFs into structured XML or TEI encoded documents.
4. **CSV, XML, TXT Files**: Formats in which the extracted data is saved. CSV files store structured data from web pages, XML files are generated by Grobid for structured document representation, and TXT files contain plain text extracted from PDFs.
5. **AWS S3**: A storage service used to save the processed files (CSV, XML, TXT) in the cloud, ensuring data durability and accessibility.
6. **Snowflake**: A cloud data platform where structured data from CSV and XML Metadata files is loaded for analysis and querying purposes.

#### **Workflow**





1. **Data Collection**: The process begins with the collection of data from the CFA Institute Website using Selenium for dynamic content interaction and Beautiful Soup for parsing HTML content. This step results in the extraction of links to PDF documents and other relevant data, which is then saved into a CSV file.
2. **PDF Text Extraction**: PDF documents for Level 1, Level 2, and Level 3 are processed using PyPdf2 and Grobid. PyPdf2 extracts text directly into TXT files, while Grobid converts PDFs into structured XML documents, which are further transformed into TXT files for uniformity.
3. **Data Storage and Processing**:
   1. The CSV file containing scraped data is uploaded to AWS S3 for storage and further processed in Snowflake for analysis.
   2. TXT files generated from PDF text extraction are also stored in AWS S3, providing a cloud-based repository for all extracted text.
   3. Metadata of the XML files produced by Grobid are loaded into Snowflake along with a link to the amazon s3

#### 

# **Web Scraping and Dataset Creation**

#### **Overview**:

This script is designed to scrape educational content from a specified set of URLs. It navigates through web pages to collect links to reading materials, extracts relevant information from each reading, and then saves this data into a CSV file. The primary focus is on gathering information such as the topic name, publication year, difficulty level, introduction, learning outcomes, summary, and links to the summary page and PDF files.

#### **Dependencies:**

1. **Selenium:** A web automation tool used to programmatically navigate through web pages and interact with web elements.
2. **WebDriver-Manager:** Helps to manage browser drivers easily, ensuring the correct version of a driver is used.
3. **BeautifulSoup4:** A library for parsing HTML and XML documents, used here to extract and manipulate data from web pages.

#### **Setup:**

1. Install the required packages using pip:

pip install selenium webdriver-manager beautifulsoup4

1. This script uses Chrome and webdriver-manager to automatically handle driver requirements.

#### 

#### **Usage:**

To run the script, simply execute it in a Python environment. The script performs the following steps:

1. Initializes a Selenium WebDriver for Chrome.

2. Iterates through a list of URLs, each pointing to a page with links to reading materials.

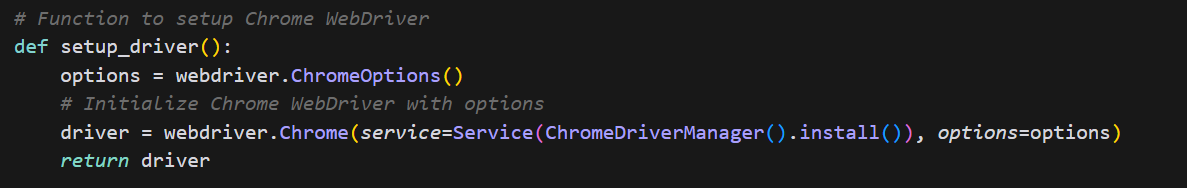
3. Collects all reading links from each page.

4. Visits each reading link to scrape relevant information.

5. Saves the collected data into a CSV file named Team05.csv.

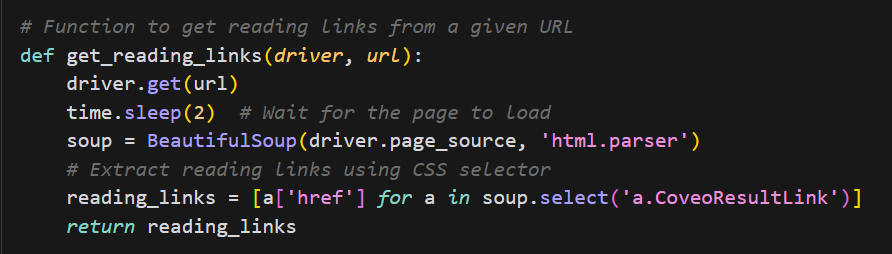
#### **Functions:**

1. ***setup\_driver():***



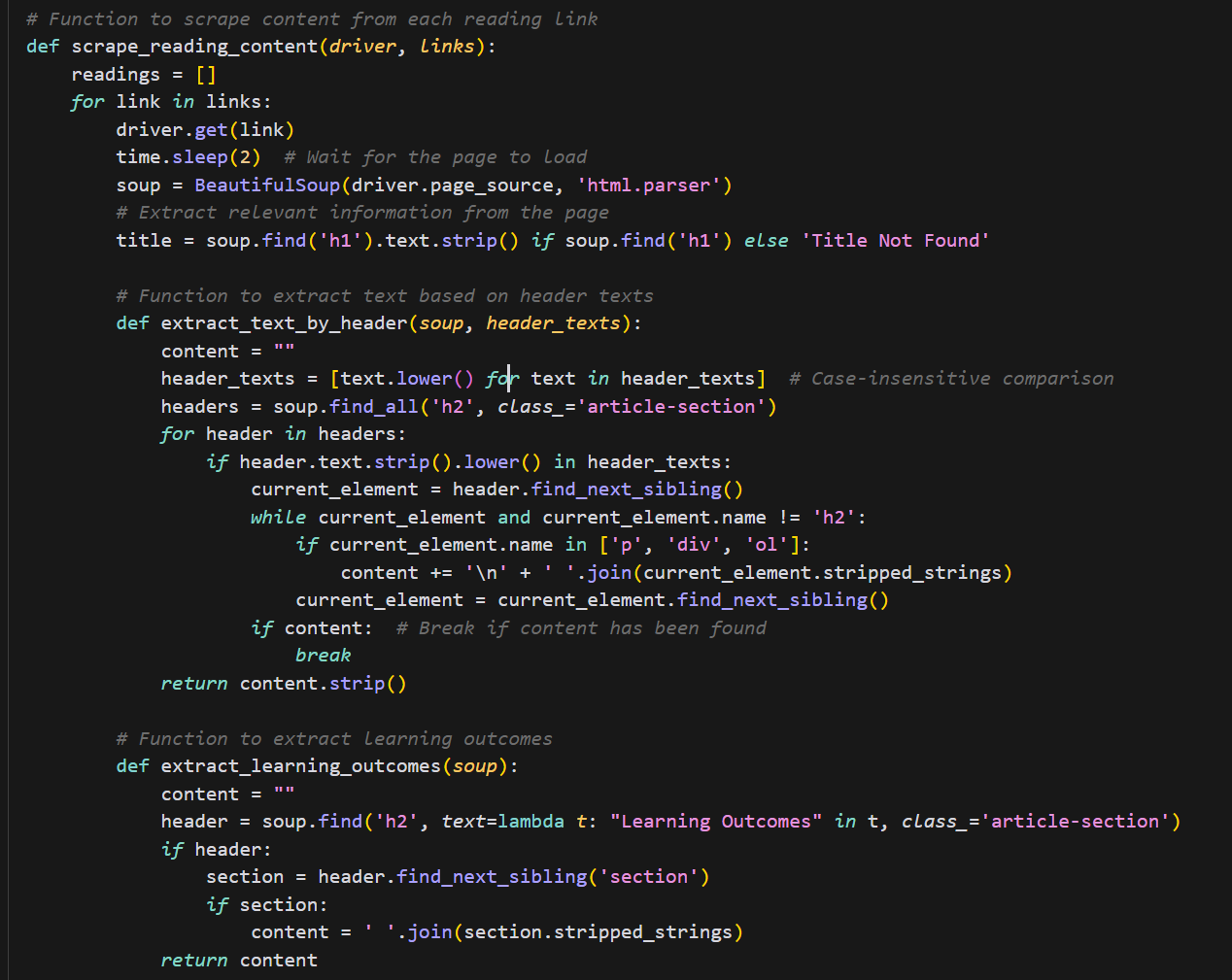
* 1. This function initializes and configures a Chrome WebDriver instance. It sets up the necessary options for the WebDriver and uses ChromeDriverManager().install() to automatically manage the driver binary required for Chrome. This simplifies the process of setting up Selenium by ensuring you always have the correct version of the driver without manually downloading it. The function then returns the configured WebDriver instance for use in navigating and interacting with web pages.

1. ***get\_reading\_links(driver, url):***



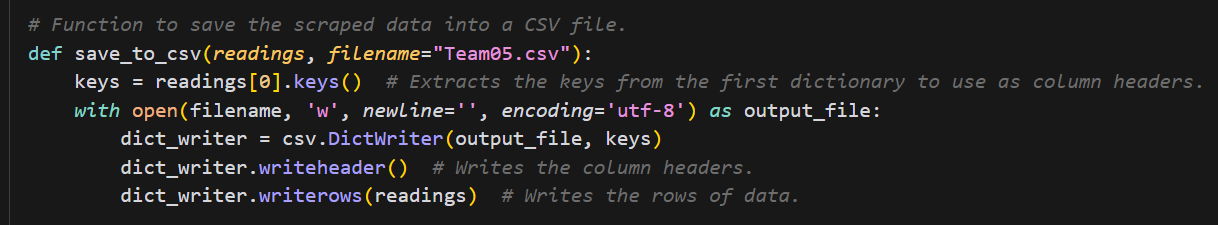
* 1. This function navigates to the specified URL and waits for the page to load. It then uses BeautifulSoup to parse the page's HTML content. The function looks for all anchor tags (<a>) with a class of CoveoResultLink, which are assumed to be the links to the reading materials on the page. It extracts the href attribute (the URL) from each of these anchor tags and returns a list of these URLs. This list represents the reading materials found on the page.

1. ***scrape\_reading\_content(driver, links):***
   1. This function takes a WebDriver instance and a list of URLs (links) to individual reading materials. For each URL in the list, it navigates to the page, waits for it to load, and then parses the page content using BeautifulSoup. It extracts various pieces of information from the page, such as the title, publication year, difficulty level, introduction, learning outcomes, summary, and links to the full PDF file if available. This is achieved through a combination of searching for specific HTML elements and classes (e.g., <h1> for the title, a span with a class of content-utility-curriculum for the year, etc.) and custom functions like extract\_text\_by\_header and extract\_learning\_outcomes that look for content under specific headers. The extracted information is then compiled into a dictionary for each reading and added to a list of readings, which is returned at the end.



### 

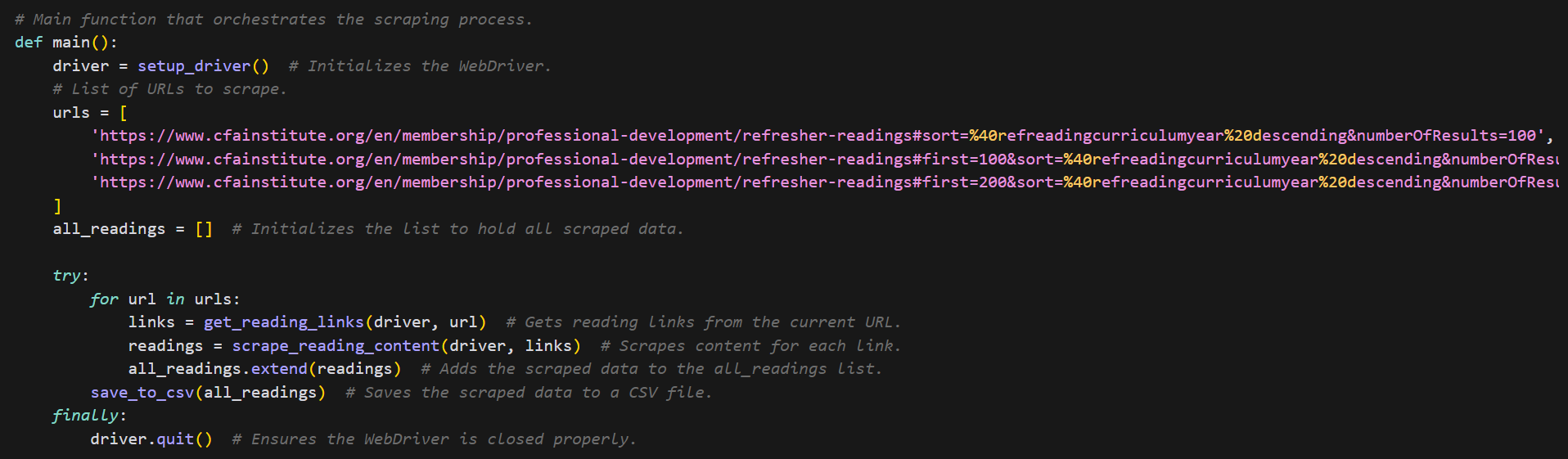
1. ***save\_to\_csv(readings, filename="Team05.csv"):***



* 1. This function takes the list of dictionaries (readings) containing the scraped data for each reading material and a filename for the CSV file. It uses Python's csv module to write this data into a CSV file. The keys from the first dictionary in the list are used as column headers. Each dictionary in the list represents a row in the CSV, with the dictionary values corresponding to the cell values under their respective headers.

### 

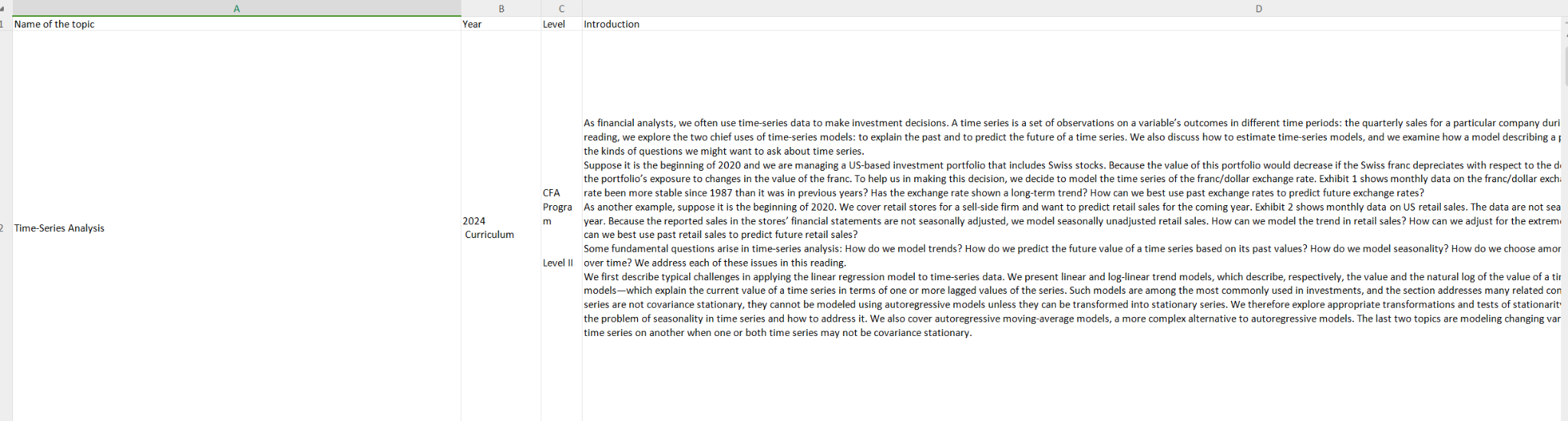
1. ***main():***



* 1. The main function orchestrates the entire scraping process. It initializes the WebDriver, defines the URLs to scrape, and iterates through these URLs to collect links to reading materials. It then scrapes the content from each link and aggregates all the scraped data. Finally, it saves this data to a CSV file and ensures the WebDriver is properly closed, regardless of whether the scraping process completes successfully or encounters an error.

**10.**

#### **Output:**



The output of the whole scrapping is stored in csv as requested in the assignment. The above image is just an example of how the output is being stored. The columns that have been generated are Name of the topic, Year, Level, Introduction, Learning Outcomes, Summary, Link to the Summary Page and Link to the PDF file

**PDF Extraction**

# **PyPDF**

## **Introduction**

This document outlines the process of extracting text from PDF files using the PyPDF2 library in Python. The guide provides a step-by-step approach to install PyPDF2, extract text from PDF files, and save the extracted text to a structured text file. This process is particularly useful for handling multiple PDF documents where consistent text extraction and structuring are required.

## **Installation**

Before extracting text from PDF files, ensure that PyPDF2 is installed in your Python environment. PyPDF2 is a pure-Python PDF library capable of splitting, merging, cropping, and transforming the pages of PDF files. It can also extract information such as text and metadata from PDFs.

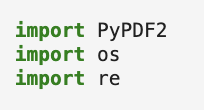


## **Extracting Text with PyPDF2**

The following Python code demonstrates how to use PyPDF2 to extract text from a list of PDF files. The extracted text is cleaned to remove undesired characters or patterns, such as multiple newlines, and saved into structured text files.

### **Import Libraries**

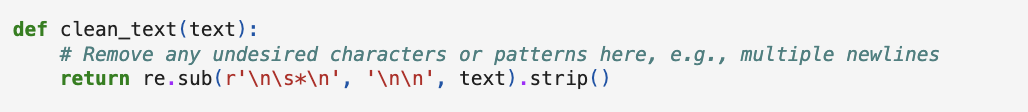
First, import the necessary libraries:



### **Define Helper Functions**

Next, define a set of helper functions to facilitate text extraction, cleaning, and saving.

Clean Text

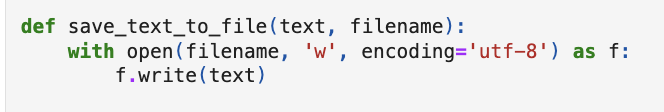


#### 

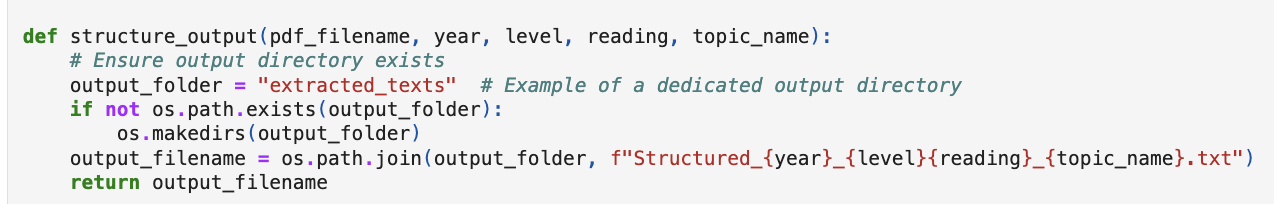
#### Extract Text from PDF



#### Save Text to File



#### Structure Output Filename



### 

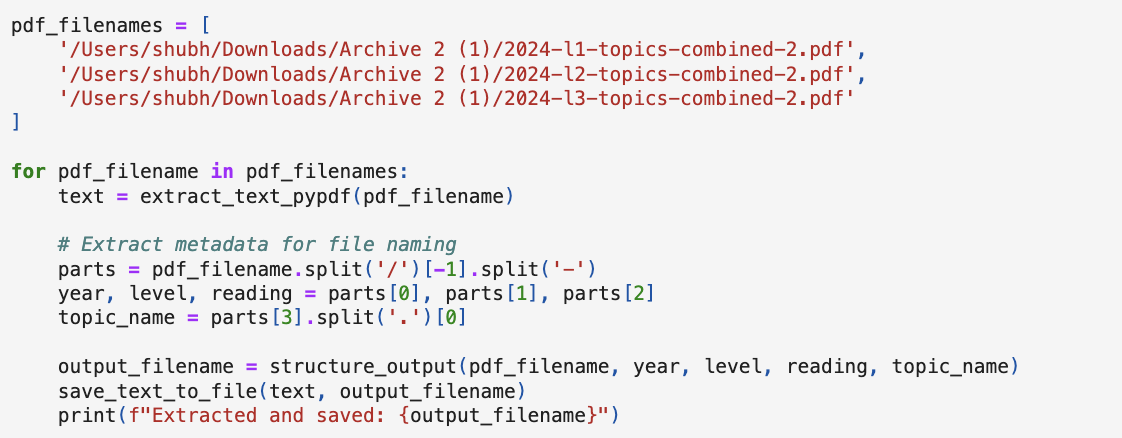
### 

### 

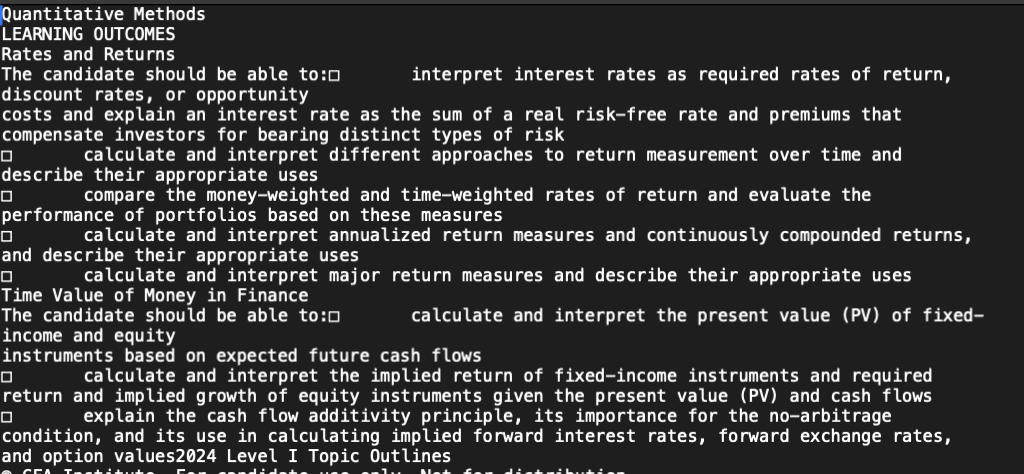
### 

### **Process PDF Files**

Specify the paths to the PDF files to be processed, extract text from each, and save the structured output.



OUTPUT TEXT SAMPLE :



# **GROBID**

## Introduction:

Grobid (GeneRation Of BIbliographic Data) is a powerful machine learning library for extracting, parsing, and structuring raw data from PDF files into XML or TEI formats. This documentation outlines the procedure to utilize Grobid via its Docker container to process PDF documents, extracting structured data for further analysis or processing.

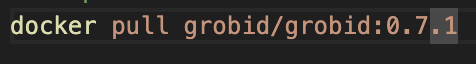
## Prerequisites

* Docker: Ensure Docker is installed and running on your machine. Docker is used to run Grobid in an isolated container, making it easy to deploy without worrying about specific dependencies.
* Curl: This command-line tool is used for transferring data with URLs and will be utilized to interact with the Grobid API.

***FOR THE WHOLE PROCESS A SHELL SCRIPT HAS BEEN IMPLEMENTED***

## Installation and Setup

1. Pull the Grobid Docker Image: Download the latest Grobid Docker image from Docker Hub.



1. Run the Grobid Container: Start the Grobid Docker container in the background. This command also maps the container's ports to the local machine, allowing local access to the Grobid service.



* -d runs the container in detached mode (in the background).
* --rm removes the container once it's stopped.
* --name grobid assigns a name to the container for easier reference.
* -p 8070:8070 -p 8071:8071 maps the container's ports to the host.

1. Initialization Wait: Grobid needs a few moments to initialize after the container starts. Adjust the wait time based on your system's performance.



## Processing PDF Files with Grobid

1. Define the base path where your PDF files are located. Use curl to send PDF files to the Grobid service for processing, saving the output in XML format.

BASEPATH="/Users/shubh/Downloads/Archive 2 (1)"

echo "Processing Level I..."

curl -v --form input=@"${BASEPATH}/2024-l1-topics-combined-2.pdf" http://localhost:8070/api/processFulltextDocument > Grobid\_RR\_2024\_LevelI\_combined.xml

Stages the process into 3 files one by one (Level II & Level III process mentioned below)

echo "Processing Level II..."

curl -v --form input=@"${BASEPATH}/2024-l2-topics-combined-2.pdf" http://localhost:8070/api/processFulltextDocument > Grobid\_RR\_2024\_LevelII\_combined.xml

echo "Processing Level III..."

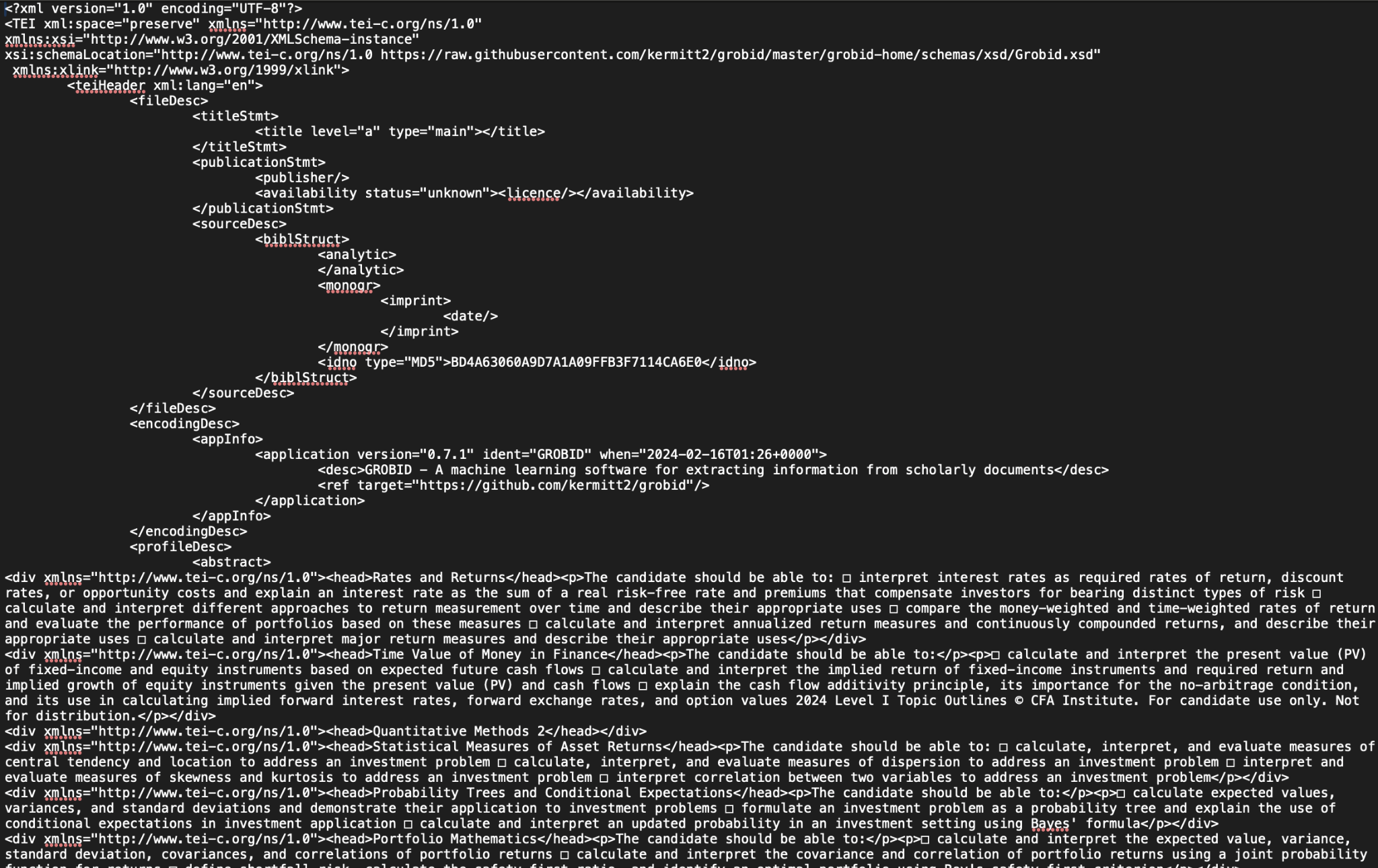
curl -v --form input=@"${BASEPATH}/2024-l3-topics-combined-2.pdf" http://localhost:8070/api/processFulltextDocument > Grobid\_RR\_2024\_LevelIII\_combined.xml

1. Stop the Docker container:

docker stop grobid

echo "Processing completed."

OUTPUT Snippet (XML file)



## **Conversion of XML to Text using LXML**

This process consists of converting XML documents into structured text using the lxml library in Python. Specifically, it focuses on XML documents adhering to the TEI (Text Encoding Initiative) format, which is frequently used for representing rich textual content in digital humanities, libraries, and scholarly publications.

Before proceeding, ensure that the lxml library is installed in your Python environment. lxml is a powerful and easy-to-use library for processing XML and HTML in Python

## **Extracting Structured Text**

The goal is to extract structured text from an XML file, preserving significant elements such as document titles, abstracts, sections, headers, and paragraphs.

#### Importing LXML

First, import the etree module from lxml, which provides the necessary functions to parse and navigate the XML document.



#### Defining the Extraction Function

The extract\_text\_with\_structure function is designed to parse a TEI XML file, extract relevant content, and format it into a structured text representation.

#### Loading and Parsing the XML

The function begins by loading and parsing the TEI XML file using etree.parse, followed by defining the TEI namespace to facilitate XPath queries. ( The **Text Encoding Initiative** (TEI) is a consortium which collectively develops and maintains a standard for the representation of texts in digital form)



#### Extracting Document Title: Using an XPath query, the document title is retrieved and formatted.



#### Extracting Abstract

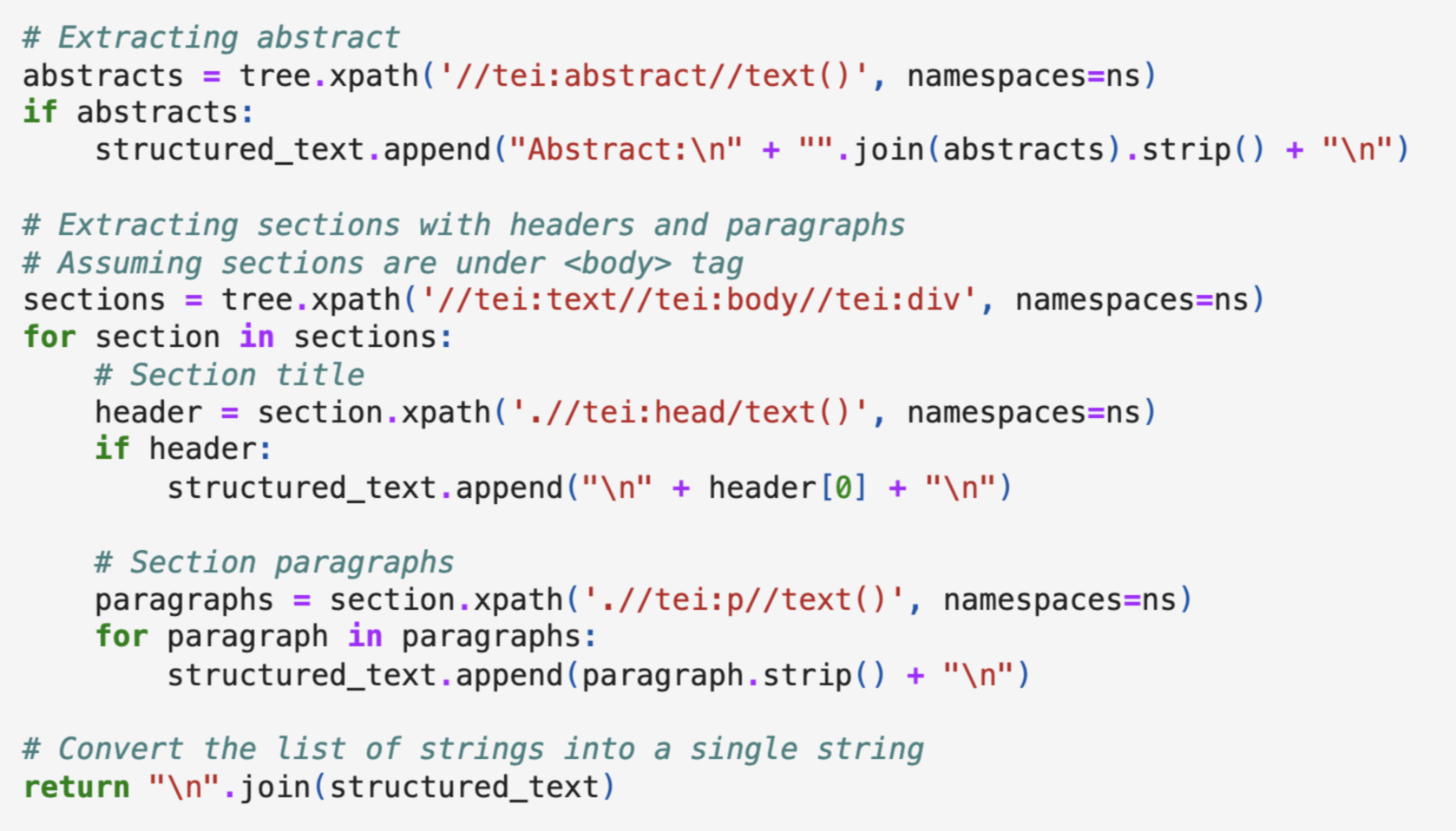
Similarly, the abstract text is extracted, if available, using XPath to navigate through the XML structure.

#### Extracting Sections, Headers, and Paragraphs

The body of the document is processed to extract sections, including their headers and paragraphs. Each section is treated as a distinct block of text.

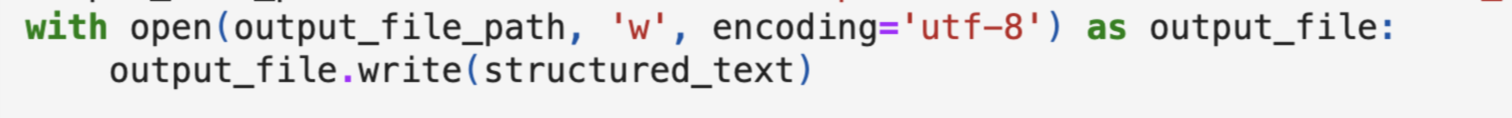
#### Formatting the Output

The extracted elements are formatted and appended to a list, which is then joined into a single string representing the structured text

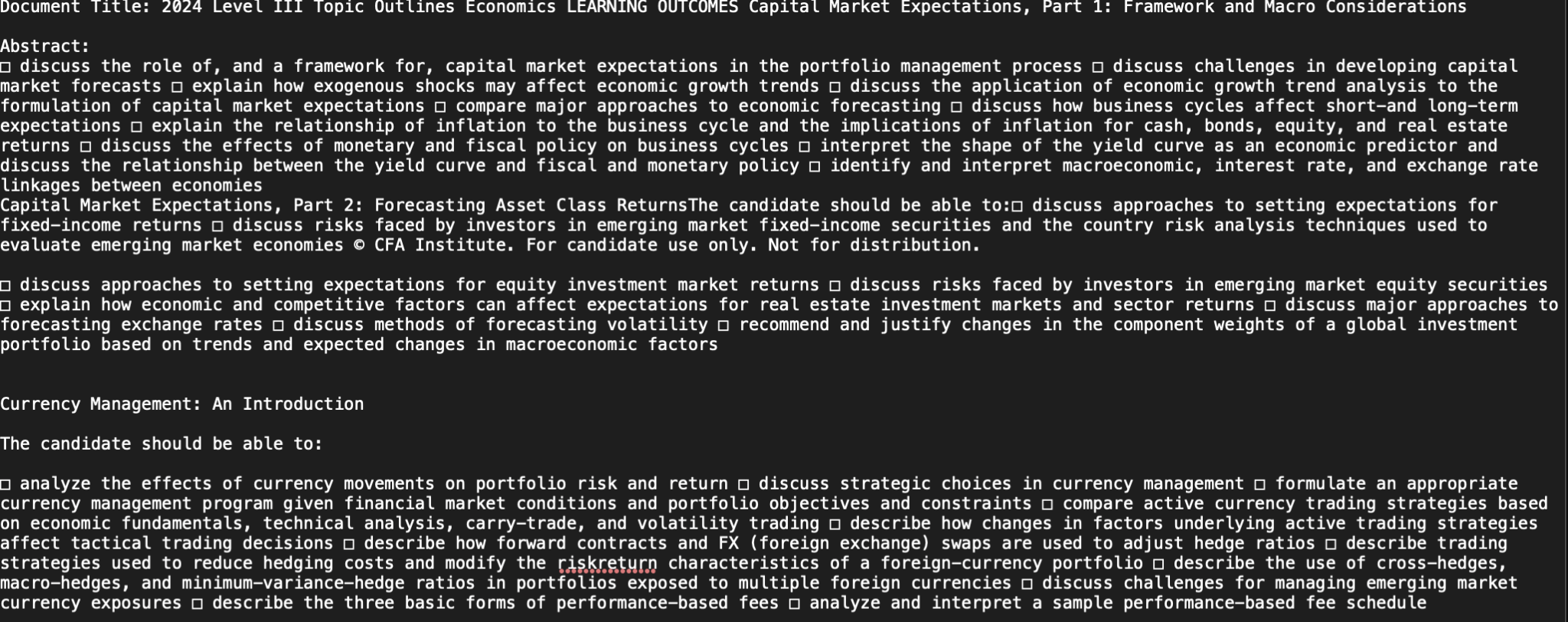


## Saving the Structured Text

## The structured text is saved to a new file, preserving the logical organization of the original XML document.



OUTPUT Snippet (Text extraction using lxml)



# **Extracting Metadata from XML using LXML**

### **Parsing TEI XML**

The process begins by parsing a TEI XML document using etree.parse() from the lxml library. The XML namespace is defined to facilitate XPath queries for extracting specific elements:



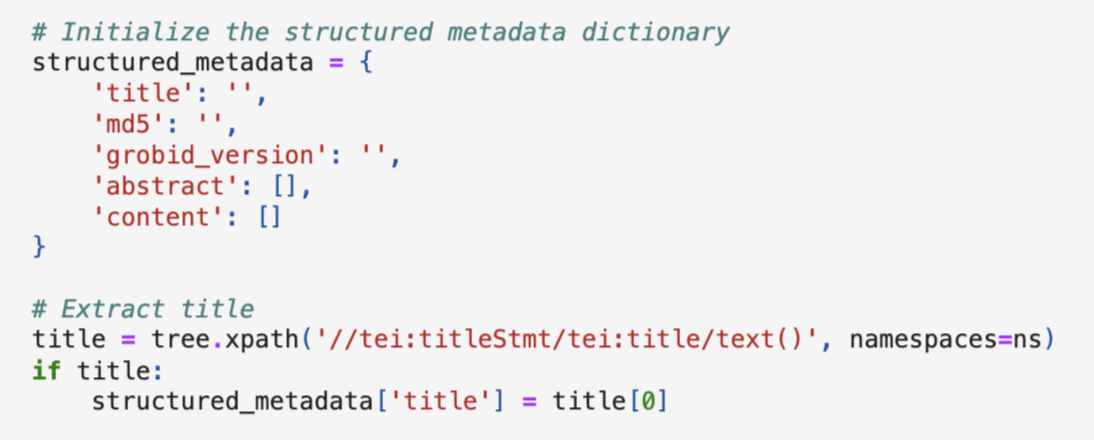
### 

### 

### 

### **Extracting Elements**

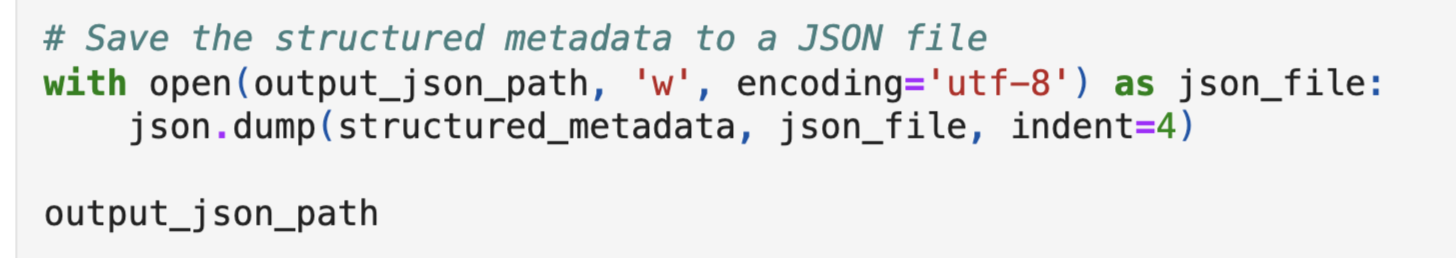
Key elements such as document title, MD5 identifier, Grobid version, abstract, and main content are extracted using XPath expressions. These elements are then stored in a structured metadata dictionary:



### Example: Extract title code given, follow similar pattern for md5, grobid\_version, abstract and content

### **Saving Metadata in JSON Format**

The structured metadata dictionary is serialized to a JSON file using the json.dump() method, providing a structured and easily accessible representation of the document's metadata



### **Merging Sections and Writing to CSV**

To accommodate scenarios where sections under the same heading should be merged, the extraction process can be adjusted to combine paragraphs for each unique heading. This combined content is then saved in a CSV file, with each row representing a heading and its associated paragraphs:

import csv

def parse\_tei\_xml\_to\_csv(file\_path, output\_csv\_path):

combined\_sections = {}

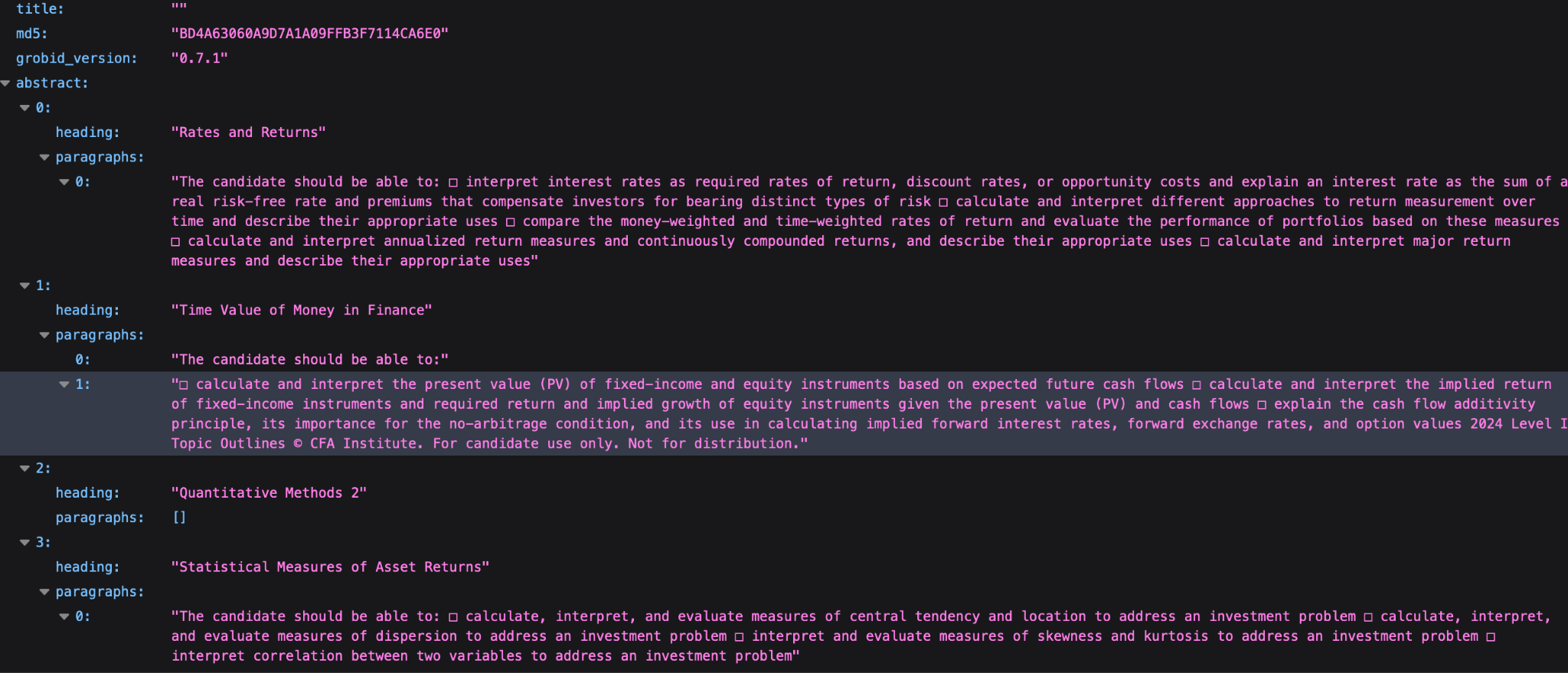
# Combine paragraphs under the same heading

# CSV writing logic follows

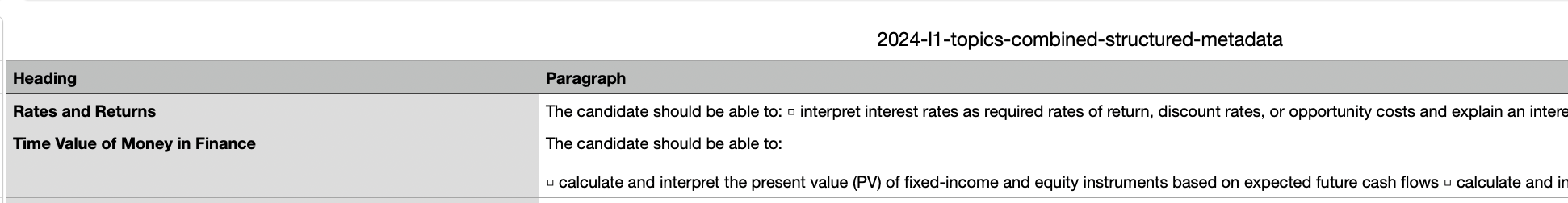
output\_csv\_path = '/path/to/output.csv'

parse\_tei\_xml\_to\_csv(xml\_file\_path, output\_csv\_path)

OUTPUT JSON Snippet



OUTPUT CSV Snippet



# **Database Upload**

**Overview**

This script is designed to automate the process of creating a database, a table, and a warehouse in Snowflake, and then uploading data from a CSV file into the Snowflake table. It utilizes SQLAlchemy for database connection and operations, ensuring a more abstract and Pythonic way of handling SQL queries.

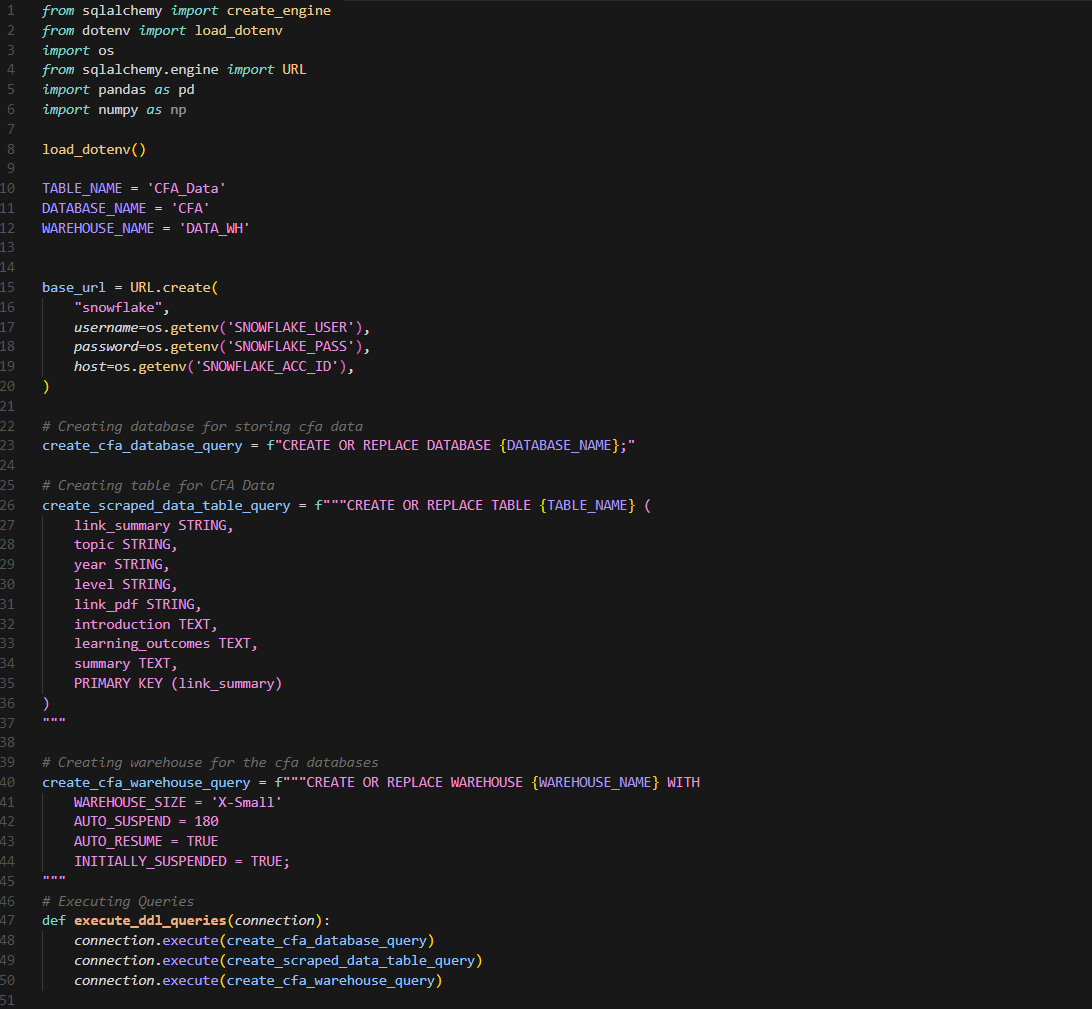
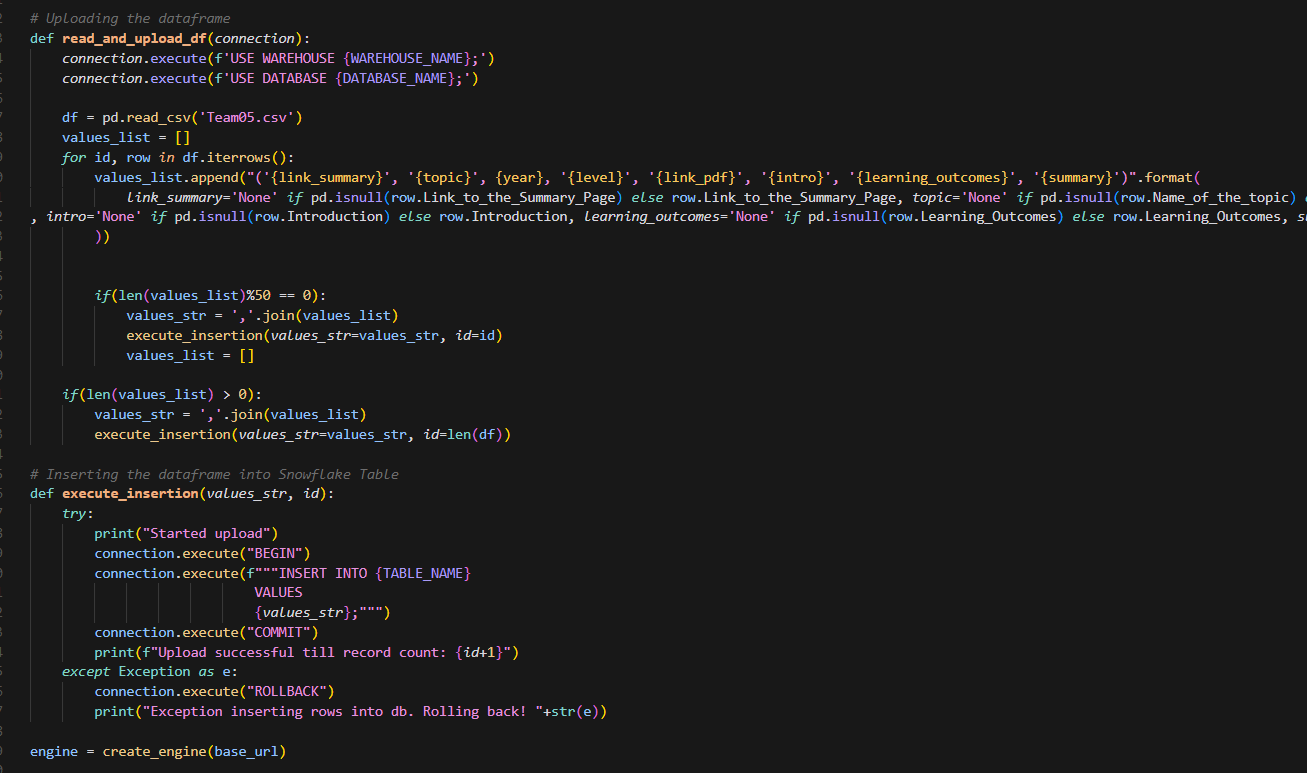
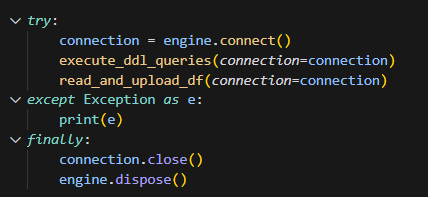
**Dependencies**

* SQLAlchemy: Used for creating engine connections and executing SQL queries in a more abstract manner.
* pandas: For reading and processing the CSV file.
* python-dotenv: For loading environment variables from a .env file, which includes Snowflake credentials.
* os: For accessing environment variables.

**Environment Setup:**

* We have a .env file in the root directory with the following variables:
  + SNOWFLAKE\_USER: Your Snowflake username.
  + SNOWFLAKE\_PASS: Your Snowflake password.
  + SNOWFLAKE\_ACC\_ID: Your Snowflake account ID.
* Place the Team05.csv file in the root directory.
* Run the script. It will automatically handle the creation of the database, table, and warehouse, and proceed with uploading the data into Snowflake.

**Steps:**

1. **Database and Table Creation**
   1. The script starts by constructing a Snowflake URL using credentials from environment variables. It then creates a SQLAlchemy engine and establishes a connection. The first operation is to create a database named CFA, a table named CFA\_Data, and a warehouse named DATA\_WH. The table is designed to store various fields related to CFA data, including a primary key.
   2. 
2. **Data Upload**
   1. The script reads data from a CSV file named Team05.csv using pandas. It processes the data, replacing any null values with 'None', and prepares it for insertion into the Snowflake table. The data upload is batched in groups of 50 records to optimize performance and manage resources efficiently.
   2. 
3. **Error Handling**
   1. The script includes basic error handling during the data insertion process. If an error occurs, it rolls back the transaction to ensure data integrity and prints an error message to the console.
4. **Cleanup**
   1. Finally, the script ensures that the database connection is closed and the engine is disposed of properly, regardless of whether the operation was successful or if an error occurred.
   2. 

**Outcome:**

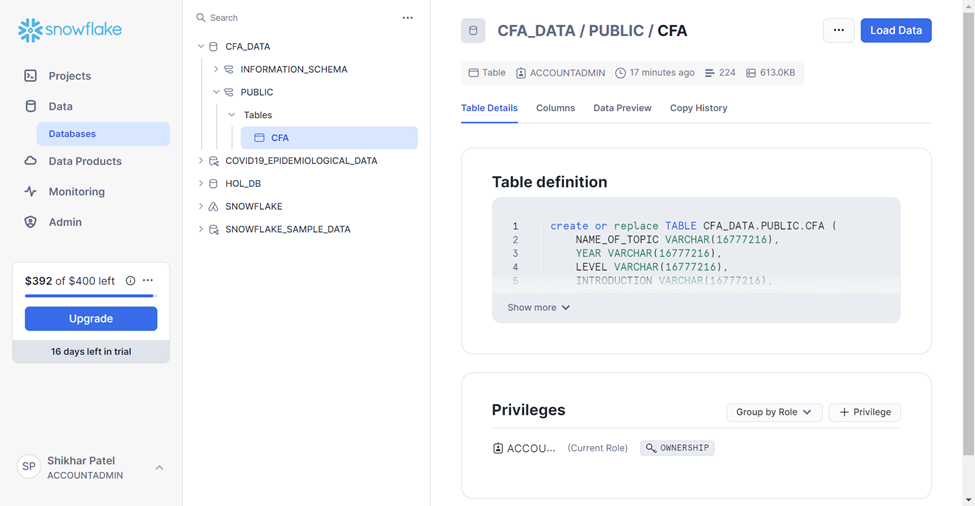
Upon successful execution of the script, the following outcomes are expected:

- Creation of a database named `CFA\_Data` in Snowflake.

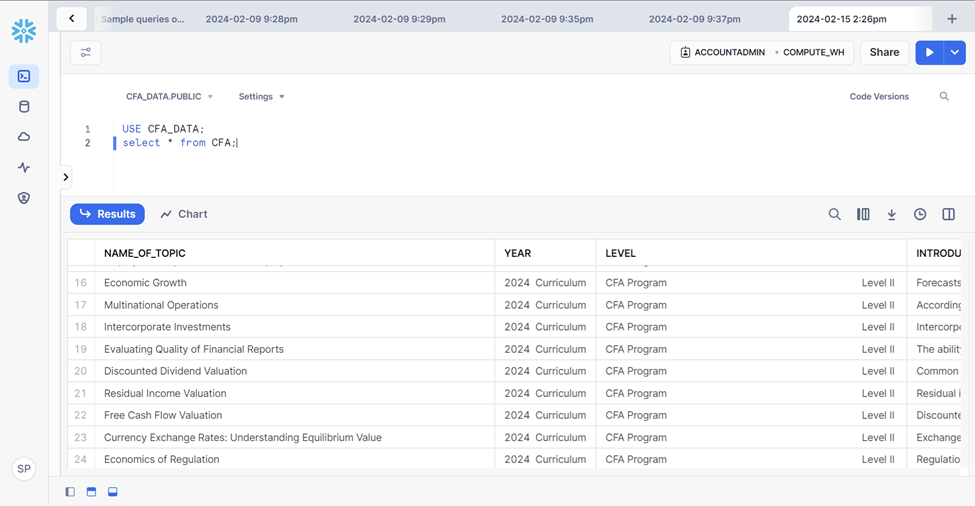
- Creation of a table named `CFA` within the `CFA\_Data` database with columns specified in the SQL query.

- Data from the CSV file (`Team05.csv`) inserted into the `CFA` table.

- If any errors occur during the process, appropriate error messages are displayed.



Executing Query to check the Data:



# **Cloud Storage Integration**

**Overview:**

In this section, we discuss a Python script designed to upload structured data (CSV) and extracted text files from Grobid and PyPDF to an AWS S3 bucket. This integration facilitates efficient cloud storage of data, enabling easy access and scalability for further analysis.

**Python Script Usage:**

The Python script utilizes the boto3 library to interact with the AWS S3 service. It defines a function named upload\_to\_s3, which takes file paths and a bucket name as input parameters. The function iterates through each file path, uploads the files to the specified S3 bucket, and returns the URLs of the uploaded files.

**Step-by-Step Explanation:**

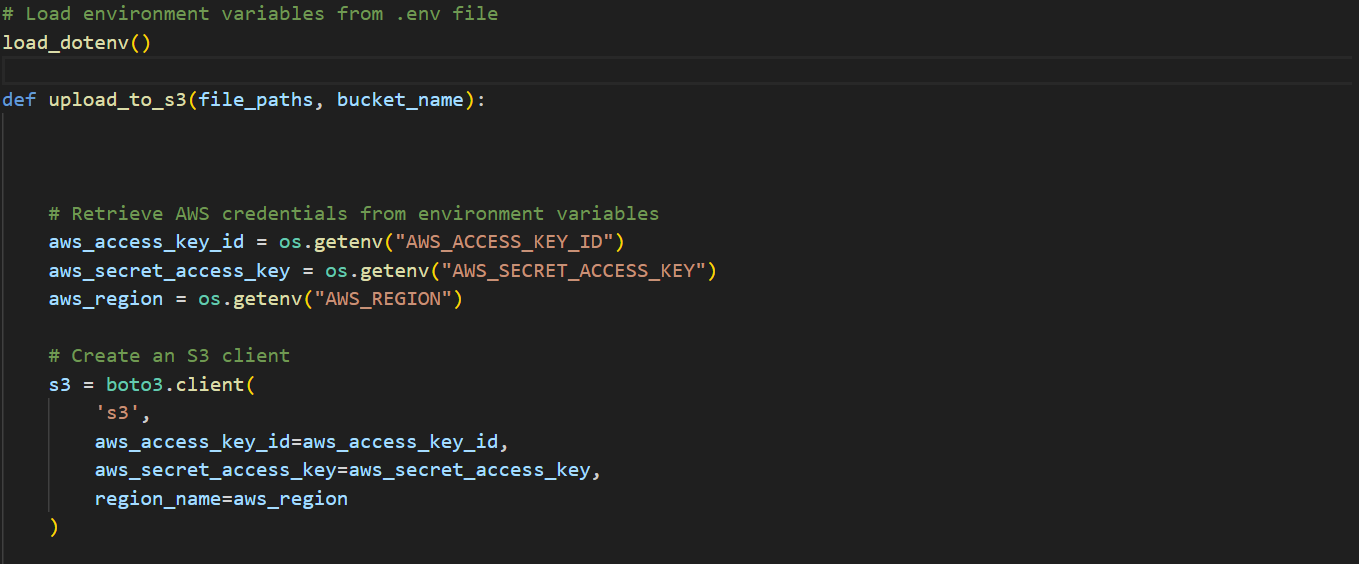
1. Import Required Libraries:

The script imports the necessary libraries, including os, boto3, and dotenv, to handle environment variables.



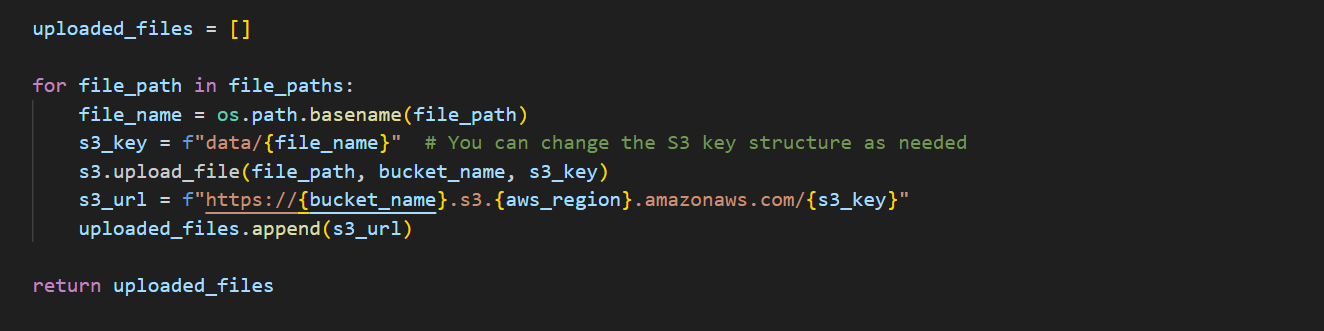
1. Load Environment Variables:

It loads environment variables from a .env file using the load\_dotenv function from the dotenv library. This ensures secure access to AWS credentials.



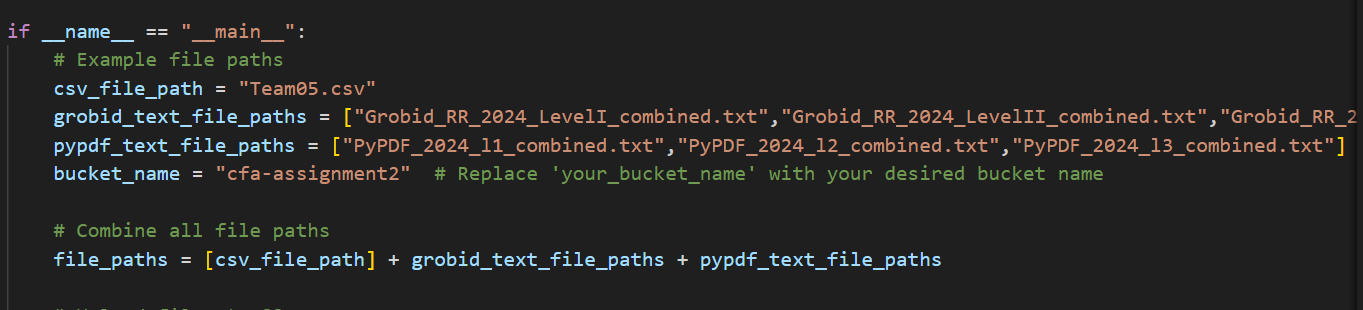
1. Define Upload Function:

The upload\_to\_s3 function is defined to handle the uploading process. It retrieves AWS credentials from environment variables and creates an S3 client using boto3.



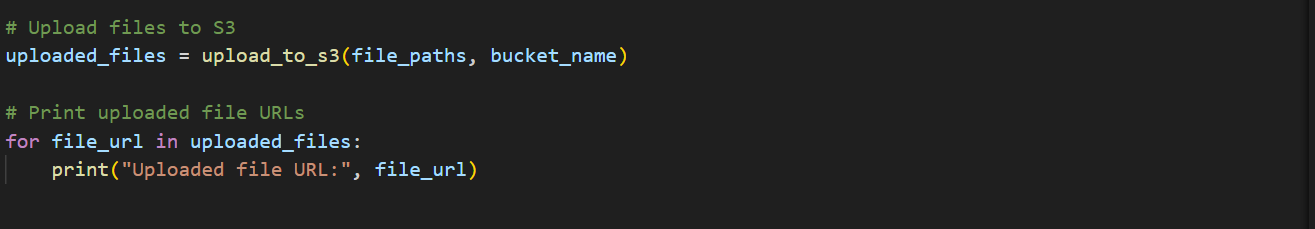
1. Iterate Through File Paths:

The script iterates through the provided file paths, which include a CSV file and text files extracted using Grobid and PyPDF.



1. Upload Files to S3 Bucket:

For each file, the script generates a unique S3 key and uploads the file to the specified S3 bucket using the upload\_file method of the S3 client.



1. Generate Uploaded File URLs:

After successful upload, the script constructs the URL of each uploaded file using the bucket name, AWS region, and S3 key.

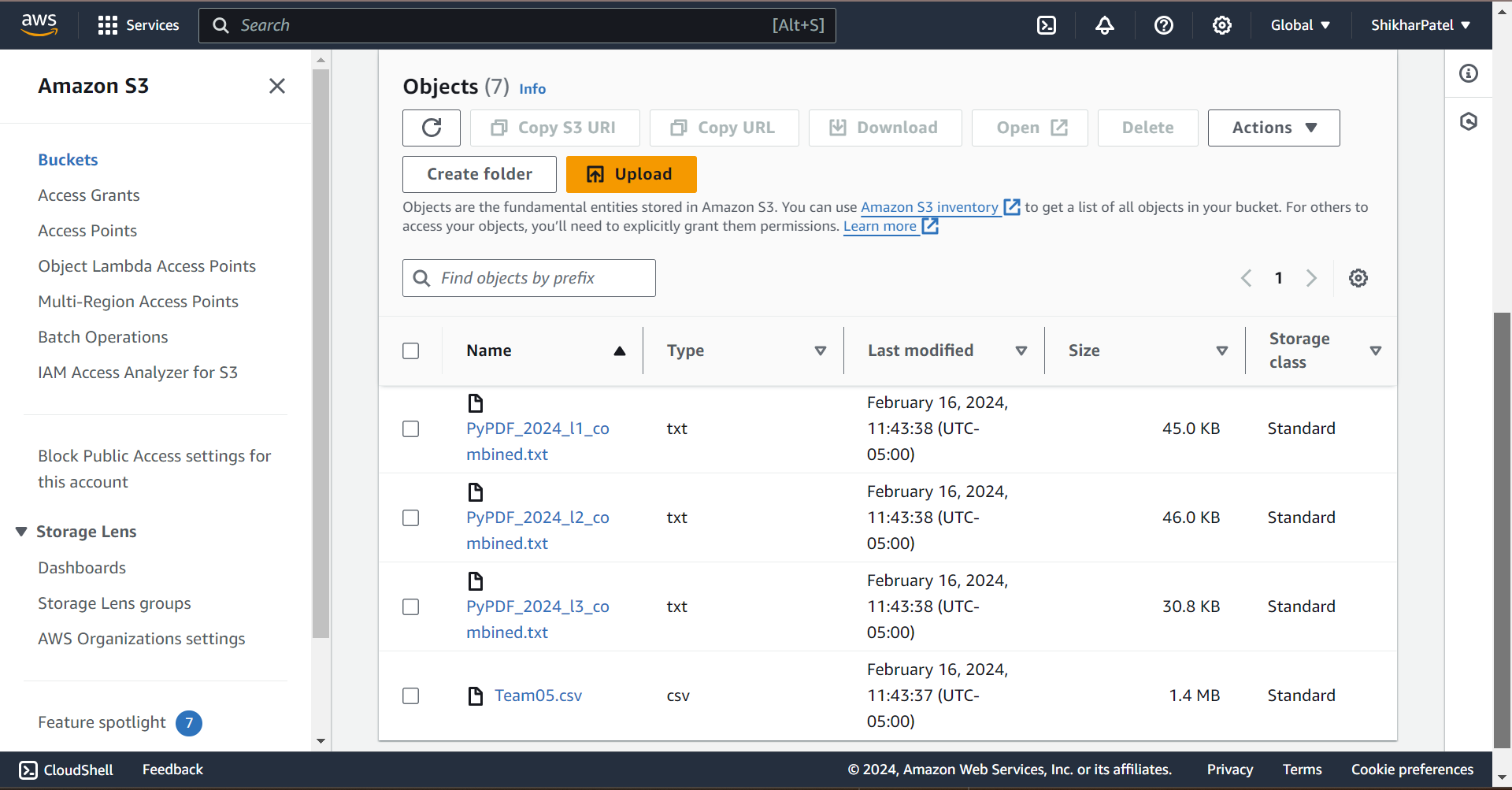
1. Display Uploaded File URLs:

Finally, the script prints the URLs of the uploaded files, providing easy access to the stored data.

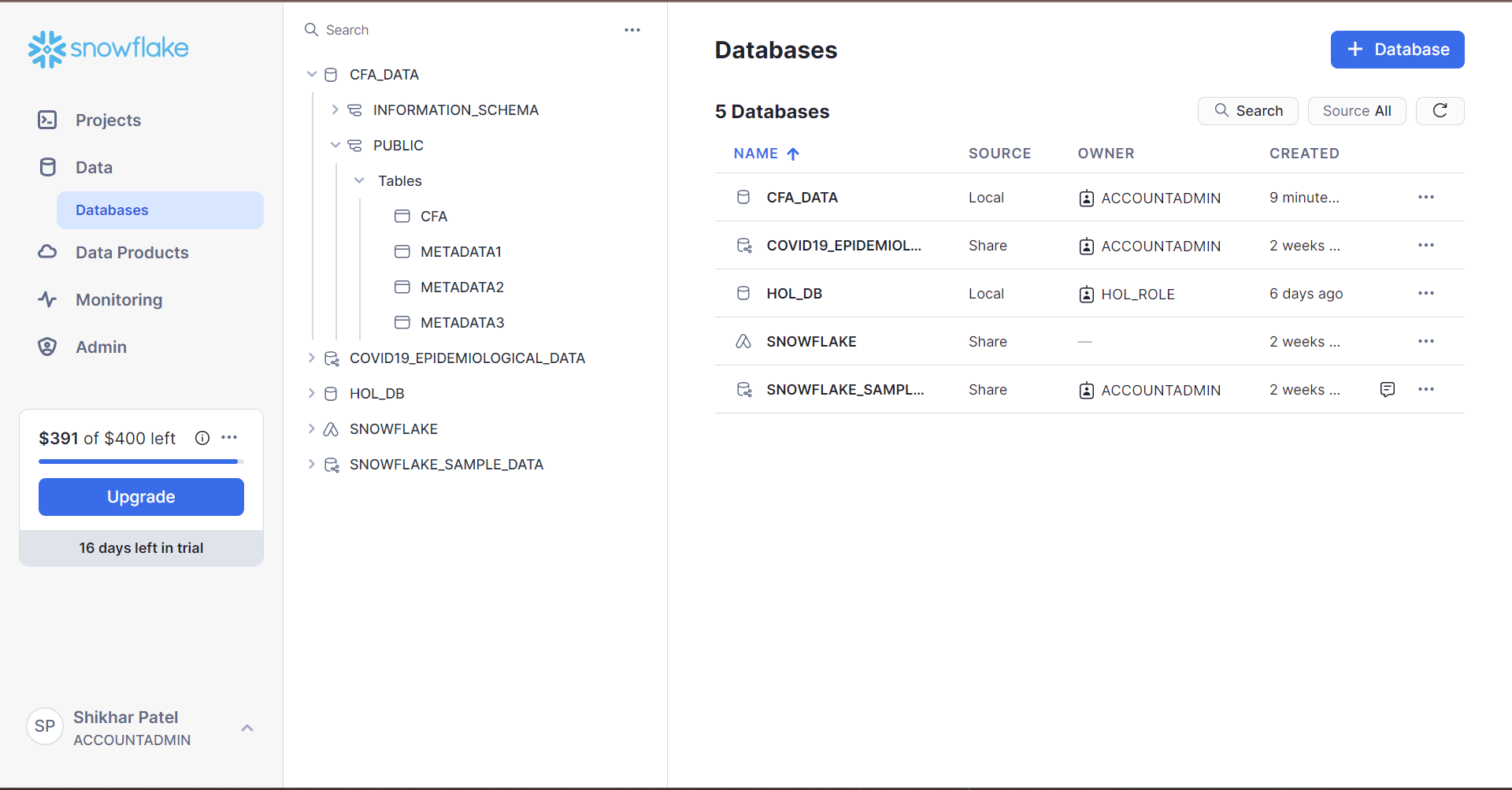
**Conclusion:**

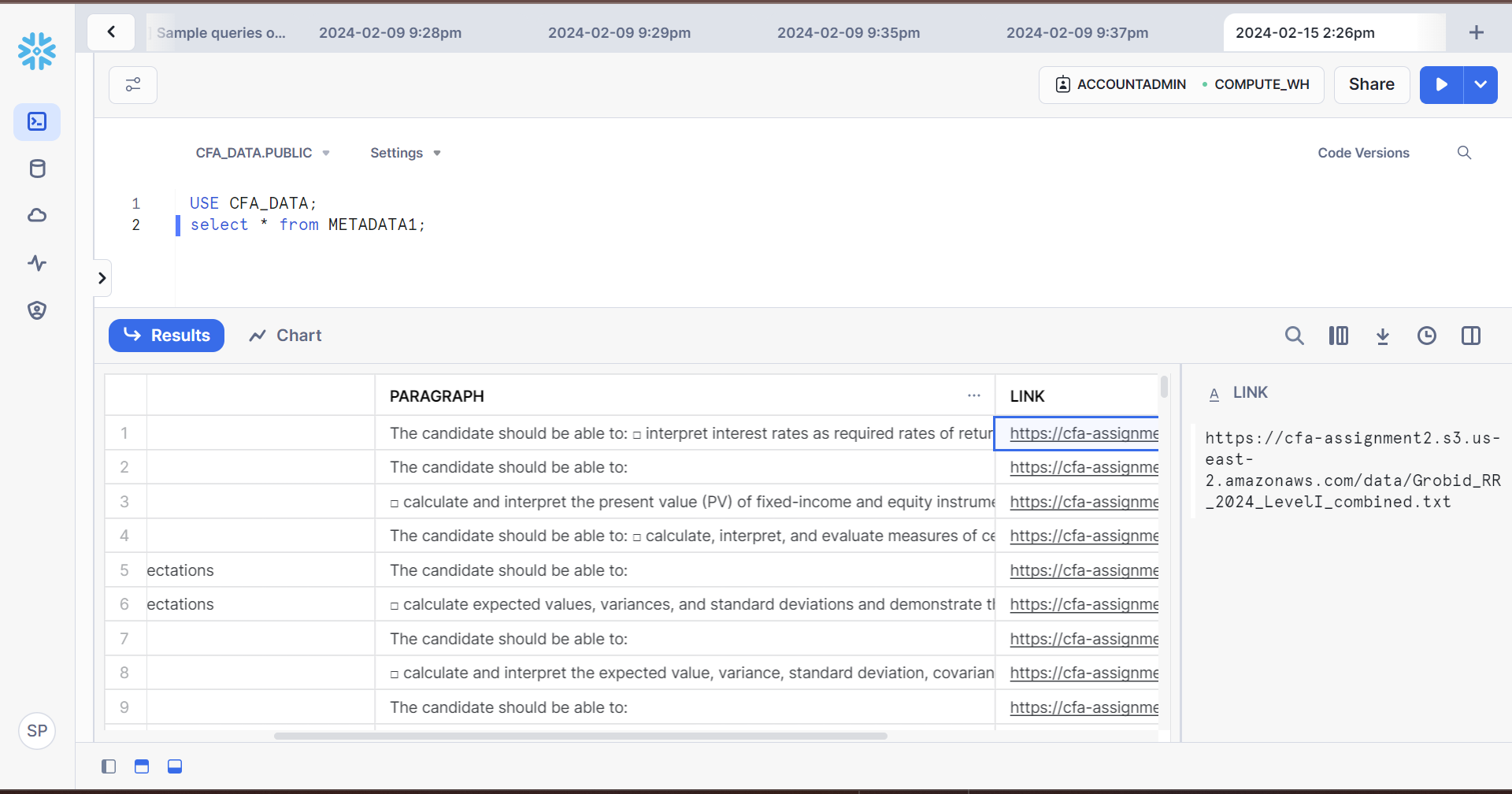
This Python script streamlines the process of uploading structured and unstructured data to an AWS S3 bucket. By leveraging cloud storage, organizations can enhance data accessibility, scalability, and security, enabling seamless integration into various analytical workflows.

The Files are uploaded successfully on S3 Bucket using the above code:



Utilizing SQLAlchemy to upload the structured metadata from step 2 (Grobid) including the link to the uploaded text file (from S3) into Snowflake database.





# 

# **References**

[*https://grobid.readthedocs.io/en/latest/Introduction/*](https://grobid.readthedocs.io/en/latest/Introduction/)

[*https://www.sqlalchemy.org/*](https://www.sqlalchemy.org/)

[*https://docs.snowflake.com/en/developer-guide/python-connector/sqlalchemy*](https://docs.snowflake.com/en/developer-guide/python-connector/sqlalchemy)

[*https://pypdf2.readthedocs.io/en/3.0.0/*](https://pypdf2.readthedocs.io/en/3.0.0/)

[*https://lxml.de/*](https://lxml.de/)

[*https://aws.amazon.com/s3/*](https://aws.amazon.com/s3/)

[*https://aws.amazon.com/iam/*](https://aws.amazon.com/iam/)