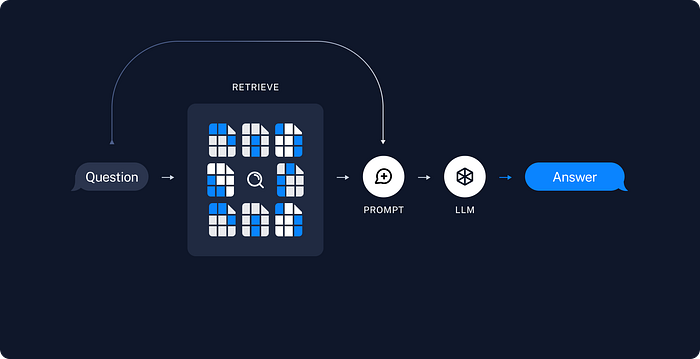
**long chain document loader medium**

**Document Loaders (Starting Point of RAG)**

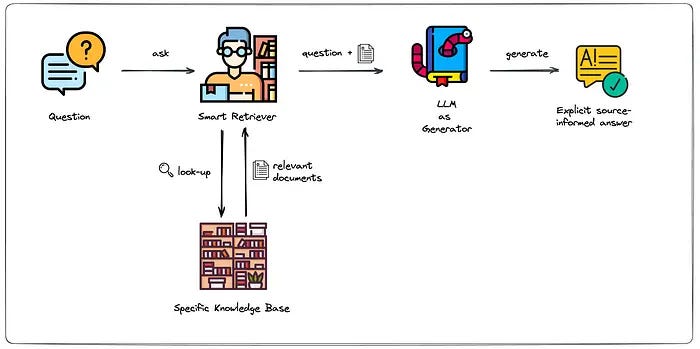


In this new series, we will explore Retrieval in Langchain — Interface with application-specific data.

**What is RAG?**

RAG (Retrieval Augmented Generation) is a framework that can be used to improve the performance of an LLM by feeding facts from external sources to refer outside its training data to generate responses. This enables improving the reliability of the generated responses and helps in handling hallucinations. By using RAG, the LLM can connect directly to updated information sources that can provide the latest information to the user.

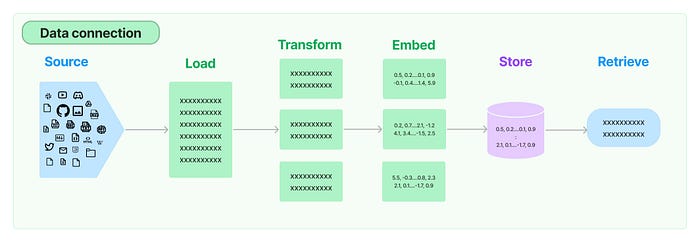
**How does RAG work?**



1. The external data is given to the LLM and stored in the form of a vector database.
2. Then the user query is converted to a vector representation and matched with the vector database.
3. Next, the RAG model augments the user input and adds the input to the relevant retrieved content.
4. The augmented prompt will allow the LLM to generate accurate responses to the user queries.

RAG has various components from loading the data to retrieving the relevant information. In this blog, we will explore the first component involved in the process of Retrieval which is ‘***Document Loaders***’.

Press enter or click to view image in full size



The very first step of retrieval is to load the external information/source which can be both structured and unstructured. Langchain provides the user with various loader options like TXT, JSON, CSV, HTML, PDF, public websites, etc. Let's see how the loaders work.

**Benefits of RAG:**

1. Use of up-to-date information
2. Better privacy
3. No limit of document size.

**Why We Need RAG?**

ChatGPT and large models are powerful, but sometimes you need to answer questions based on **your own data** —  
like policy documents, customer profiles, or proprietary knowledge —  
this is where RAG comes in.  
Instead of solely depending on the LLM’s knowledge, we augment its answers by retrieving relevant documents from an external datastore.

**RAG Components**

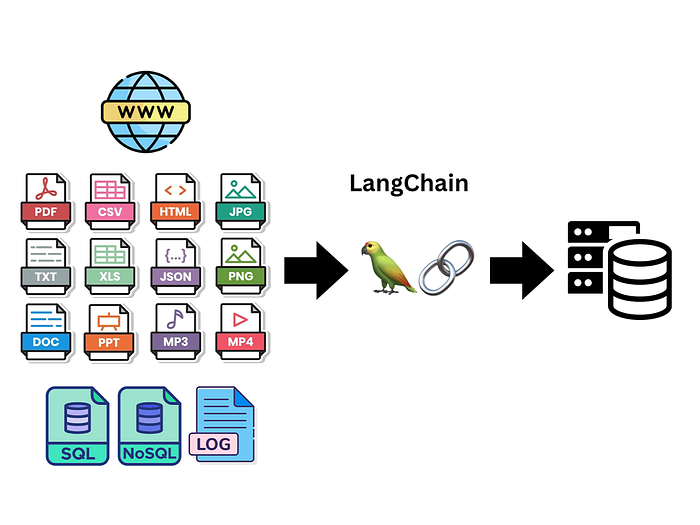
1. Document Loaders
2. Text Splitters
3. Vector Databases
4. Retrievers

In this blog we’ll deep dive into Document Loaders.

**What Are Document loaders?**

Document loaders are components in LangChain used to load data from various sources into standardized format (usually as Document objects), which can then be used for chunking, embedding, retrieval, and generation.

Press enter or click to view image in full size



**Types of Document Loaders**

1. TextLoader
2. PyPDFLoader
3. WebBaseLoader
4. CSVLoader

**TextLoader**

**TextLoader** is a simple and commonly used document loader in LangChain that reads plain text (.txt) files and converts them into LangChain Document objects.

Whenever we use any document loader we’ll get list of document object in return and each document object have 2 things first page\_content and second is metadata

**🔹 Use Case:**

✅ Ideal for loading:

* Chat logs
* Scraped text
* Transcripts
* Code snippets
* Any plain text data into a LangChain pipeline

**🔹 Limitation:**

❕ **Works only with .txt files.**

Code Example:

from langchain\_community.document\_loaders import TextLoader  
from langchain\_ollama import OllamaLLM  
from langchain\_core.output\_parsers import StrOutputParser  
from langchain\_core.prompts import PromptTemplate  
from dotenv import load\_dotenv  
  
load\_dotenv()  
  
model = OllamaLLM(model="mistral")  
  
prompt = PromptTemplate(  
 template='Write a summary for the following poem - \n {poem}',  
 input\_variables=['poem']  
)  
  
parser = StrOutputParser()  
  
loader = TextLoader('cricket.txt', encoding='utf-8')  
  
docs = loader.load()  
  
print(type(docs))  
  
print(len(docs))  
  
print(docs[0].page\_content)  
  
print(docs[0].metadata)  
  
chain = prompt | model | parser  
  
print(chain.invoke({'poem':docs[0].page\_content}))

**📁 PyPDFLoader**

**PyPDFLoader** is a document loader in LangChain used to load content from PDF files and convert each page into a Document object.

**🔹 Example:**

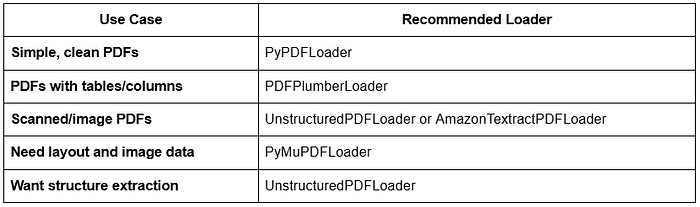
[   
 Document(page\_content='Text from page 1', metadata={"page": 0, "source": "file.pdf"}),  
 Document(page\_content='Text from page 2', metadata={"page": 1, "source": "file.pdf"}),   
 ...  
]

**🔹Limitation:**

❕ It uses the PyPDF library under the hood — **not great with scanned PDFs or complex layouts.**

**📁 PDF Loading — Use Cases and Recommended Loader**

Press enter or click to view image in full size



Code Example:

from langchain\_community.document\_loaders import PyPDFLoader  
  
loader = PyPDFLoader('dl-curriculum.pdf')  
  
docs = loader.load()  
  
print(len(docs))  
  
print(docs[0].page\_content)  
print(docs[1].metadata)

**Example-2**

from langchain.document\_loaders import PyPDFLoader  
  
loader = PyPDFLoader('../datasets/harry\_potter\_pdf/hpmor-trade-classic.pdf')  
data = loader.load()  
data

[Document(page\_content='Harry Potter and the Methods of Rationality', metadata={'source': '../datasets/harry\_potter\_pdf/hpmor-trade-classic.pdf', 'page': 0}),  
Document(page\_content='', metadata={'source': '../datasets/harry\_potter\_pdf/hpmor-trade-classic.pdf', 'page': 1}), ...

The PDF loader utilizes different sources like PyPDFium2Loader, PDFMinerLoader, PDFMinerPDFasHTMLLoader, PyMuPDFLoader, and PyPDFDirectoryLoader.

Example-3

This is used to load content from .pdf files. There are different classes of pdf loader depending on the data present in the file.

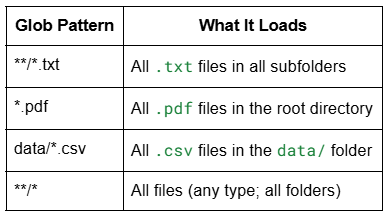
* Simple, clean PDFs: Use PyPDFLoader
* PDFs with tables/columns: Use PDFPlumberLoader
* Scanned/image PDFs: Use UnstructuredPDFLoader or AmazonTextractPDFLoader
* Need layout and image data: Use PyMuPDFLoader
* Want best structure extraction: Use UnstructuredPDFLoader

We will use PyPDFLoader in the example below.

from langchain\_community.document\_loaders import PyPDFLoader  
  
loader = PyPDFLoader("AI\_Introduction.pdf")  
documents = loader.load()  
  
print(f"Actual Data:\n{documents[0].page\_content}\n")  
print(f"Meta Data:\n{documents[0].metadata}\n")

**📁 DirectoryLoader**

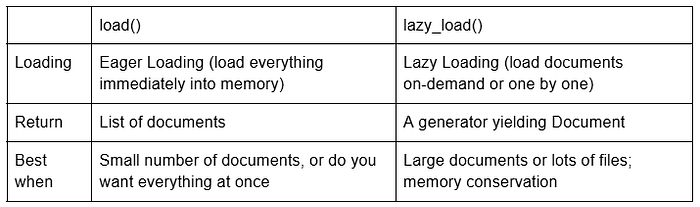
**DirectoryLoader** is a document loader in LangChain that lets you **load multiple documents from a directory (folder) of files**.



➥ \*\* means recursive search through subdirectories.

**🔹 Loading vs Lazy Loading:**

Press enter or click to view image in full size



**If we have a lot document and we have less memory we can use lazy load.**

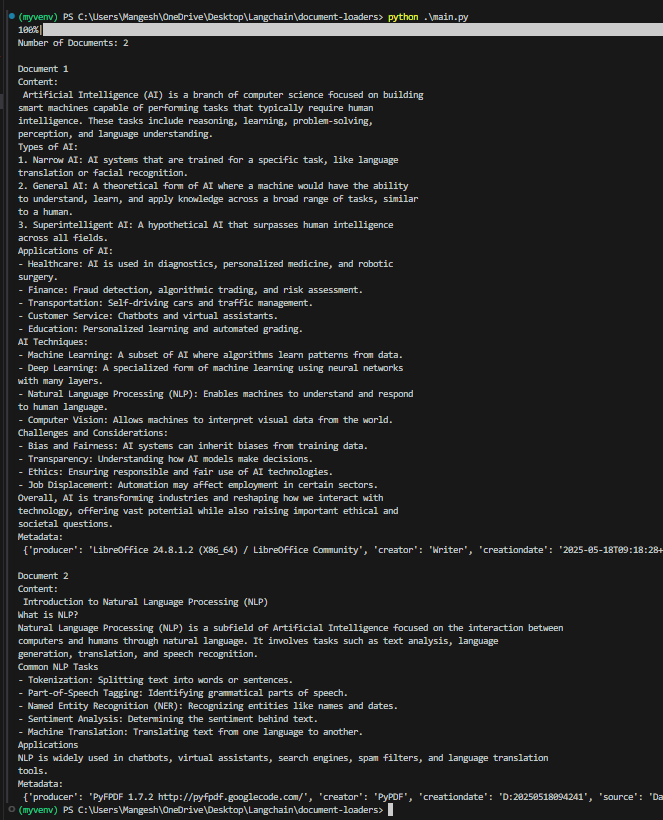
from langchain\_community.document\_loaders import DirectoryLoader, PyPDFLoader  
  
loader = DirectoryLoader(  
 path='books',  
 glob='\*.pdf',  
 loader\_cls=PyPDFLoader  
)  
  
docs = loader.load()  
# docs = loader.lazy\_load()  
  
print(docs[326])  
# for document in docs:  
# print(document.metadata)

**Example-2**

Directory loader helps in loading data from multiple files stored in a specific directory.

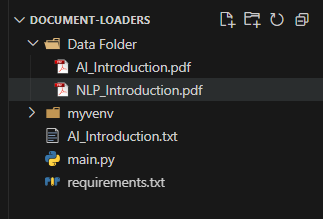
DirectoryLoader loads files from a folder using UnstructuredLoader by default, which supports formats like PDF, HTML, and Markdown. We can use the glob parameter to include specific file types—e.g., load only .pdf files while skipping .csv and .txt.

from langchain\_community.document\_loaders import DirectoryLoader  
from langchain\_community.document\_loaders import PyPDFLoader  
  
loader = DirectoryLoader("Data Folder/", glob="\*\*/\*.pdf", loader\_cls=PyPDFLoader, show\_progress=True)  
documents = loader.load()  
  
print(f"Number of Documents: {len(documents)}")  
  
for idx, document in enumerate(documents, start=1):  
 print(f"\nDocument {idx}")  
 print("Content:\n", document.page\_content)  
 print("Metadata:\n", document.metadata)



Output of Document Loader

The above code loads documents in the directory Data Folder .



Directory Structure

**📁 WebBaseLoader**

**WebBaseLoader** is a document loader in LangChain used to **load and extract text content from web pages (URLs)**.

➥ It parses HTML and converts visible text into Document.

➥ It uses **BeautifulSoup** under the hood.

**🔹 When to use:**

✅ **For blogs, news articles, or public websites where the content is primarily text-based and static.**

**🔹 Limitations:**

❕ It **can’t handle JavaScript-heavy pages well** (use SeleniumURLLoader for those).

❕ It **only parses static content** (the HTML you get immediately, not what the page renders afterwards).

Code Example:

from langchain\_community.document\_loaders import WebBaseLoader  
from langchain\_ollama import OllamaLLM  
from langchain\_core.output\_parsers import StrOutputParser  
from langchain\_core.prompts import PromptTemplate  
  
model = OllamaLLM(model="mistral")  
  
prompt = PromptTemplate(  
 template='Answer the following question \n {question} from the following text - \n {text}',  
 input\_variables=['question','text']  
)  
  
parser = StrOutputParser()  
  
url = 'https://www.flipkart.com/apple-macbook-air-m2-16-gb-256-gb-ssd-macos-sequoia-mc7x4hn-a/p/itmdc5308fa78421'  
loader = WebBaseLoader(url)  
  
docs = loader.load()  
  
  
chain = prompt | model | parser  
  
print(chain.invoke({'question':'What is the prodcut that we are talking about?', 'text':docs[0].page\_content}))

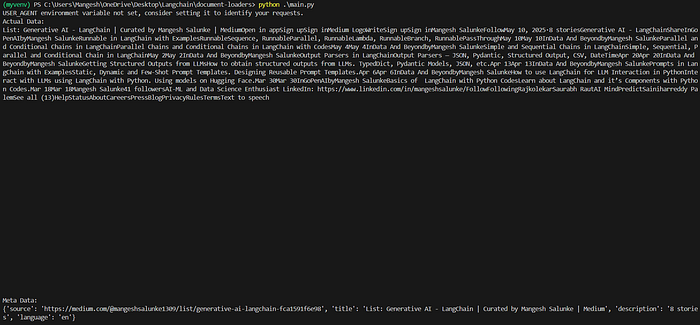
**Example-2**

Web Page Loader is used to load content from the website. It uses requests and BeautifulSoup4 python libraries inthe backend. We simply need to pass the URL of the website.

In the example below, I am loading this website: <https://medium.com/@mangeshsalunke1309/list/generative-ai-langchain-fca1591f6e98>

from langchain\_community.document\_loaders import WebBaseLoader  
  
URL = "https://medium.com/@mangeshsalunke1309/list/generative-ai-langchain-fca1591f6e98"  
  
loader = WebBaseLoader(URL)  
documents = loader.load()  
  
# print(documents)  
print(f"Actual Data:\n{documents[0].page\_content}")  
print(f"Meta Data:\n{documents[0].metadata}\n")

Press enter or click to view image in full size



Output of Web Page Loader

Visit the URL above and see the content on it. You will realize the following limitations of WebBaseLoader class:

* Only static pages are loaded. (Since BeautifulSoup is used). JavaScript-rendered content is not loaded.
* HTML elements like tables, headings, lists, and sections are often flattened into plain text. See in the output image, the output is a simple plain text.

You can read about [other Web Page Loader classes to experiment](https://python.langchain.com/docs/how_to/document_loader_web/) with other ways.

**📁 CSVLoader**

**CSVLoader** is a document loader used to **load CSV files into LangChain Document objects — one per row by default.**

It create a separate document for each row and if we have large csv file we can use lazy load.

Code Example:

from langchain\_community.document\_loaders import CSVLoader  
  
loader = CSVLoader(file\_path='Social\_Network\_Ads.csv')  
  
docs = loader.load()  
  
print(len(docs))  
print(docs[1])

**Example-2**

from langchain.document\_loaders import CSVLoader  
  
loader = CSVLoader(file\_path='../datasets/sns\_datasets/titanic.csv') # Lazy Loader  
data = loader.load()  
  
data

> [Document(page\_content='survived: 0\npclass: 3\nsex: male\nage: 22.0\nsibsp: 1\nparch: 0\nfare: 7.25\nembarked: S\nclass: Third\nwho: man\nadult\_male: True\ndeck: \nembark\_town: Southampton\nalive: no\nalone: False', metadata={'source': '../datasets/sns\_datasets/titanic.csv', 'row': 0}), Document(page\_content='survived: 1\npclass: 1\nsex: female\nage: 38.0\nsibsp: 1\nparch: 0\nfare: 71.2833\nembarked: C\nclass: First\nwho: woman\nadult\_male: False\ndeck: C\nembark\_town: Cherbourg\nalive: yes\nalone: False', metadata={'source': '../datasets/sns\_datasets/titanic.csv', 'row': 1}), ...]

The loader creates a separate document for each of the rows in the CSV.

# Getting the content of the document  
print(data[0].page\_content)   
  
# Getting the metadata  
print(data[0].metadata)

# content of the document  
survived: 0  
pclass: 3  
sex: male  
age: 22.0  
sibsp: 1  
parch: 0  
fare: 7.25  
embarked: S  
class: Third  
who: man  
adult\_male: True  
deck:   
embark\_town: Southampton  
alive: no  
alone: False  
  
#metadata information   
> {'source': '../datasets/sns\_datasets/titanic.csv', 'row': 0}

Using *source\_column, the*user can mention a specific column and pass it to the loader.

2.**HTML Loader:**

from langchain.document\_loaders import UnstructuredHTMLLoader  
loader = UnstructuredHTMLLoader('../datasets/harry\_potter\_html/001.htm')  
data = loader.load()  
data

> [Document(page\_content='A Day of Very Low Probability\n\nBeneath the moonlight glints a tiny fragment of silver, a fraction of a line…\n\n ...

3. **Markdown Loader:**

from langchain.document\_loaders import UnstructuredMarkdownLoader  
  
md\_filepath = "../datasets/harry\_potter\_md/001.md"  
  
loader = UnstructuredMarkdownLoader(file\_path=md\_filepath)  
data = loader.load()  
data

> [Document(page\_content='A Day of Very Low Probability\n\nBeneath the moonlight glints a tiny fragment of silver ...

5. **Wikipedia Loader:**

from langchain.document\_loaders import WikipediaLoader  
  
loader = WikipediaLoader(query='India', load\_max\_docs=1)  
data = loader.load()  
data

> [Document(page\_content="India, officially the Republic of India (ISO: Bhārat Gaṇarājya), ...

6. **ArXiv Loader:**

from langchain\_community.document\_loaders import ArxivLoader  
  
loader = ArxivLoader(query='2201.03916', load\_max\_docs=1) # AutoRL paper (article ID -> 2201.03916)  
data = loader.load()  
data

> [Document(page\_content='Journal of Artiﬁcial Intelligence Research 74 (2022) ...

7. **Youtube Loader:**

from langchain.document\_loaders.generic import GenericLoader  
from langchain.document\_loaders.parsers import OpenAIWhisperParser  
from langchain.document\_loaders.blob\_loaders.youtube\_audio import YoutubeAudioLoader  
  
url="https://www.youtube.com/watch?v=jGwO\_UgTS7I"  
save\_dir="docs/youtube/"  
loader = GenericLoader(YoutubeAudioLoader([url],save\_dir),  
 OpenAIWhisperParser())  
  
data = loader.load()  
data[0]

Let’s pass the retrieved information to the LLM

from langchain\_openai import ChatOpenAI  
from langchain.globals import set\_llm\_cache  
from langchain.cache import InMemoryCache  
  
  
chat = ChatOpenAI()  
set\_llm\_cache(InMemoryCache())

# Setting up the prompt templates  
  
from langchain.prompts.chat import ChatPromptTemplate, SystemMessagePromptTemplate, HumanMessagePromptTemplate, AIMessagePromptTemplate  
  
system\_template = "You are Peer Reviewer"  
human\_template = "Read the paper with the title: '{title}'\n\nAnd Content: {content} and critically list down all the issues in the paper"  
  
system\_message\_prompt = SystemMessagePromptTemplate.from\_template(system\_template)  
human\_message\_prompt = HumanMessagePromptTemplate.from\_template(human\_template)  
  
chat\_prompt = ChatPromptTemplate.from\_messages(messages=[system\_message\_prompt, human\_message\_prompt])  
prompt = chat\_prompt.format\_prompt(title=data[0].metadata['Title'], content=data[0].page\_content)  
messages = prompt.to\_messages()  
  
response = chat(messages=messages)  
  
print(response.content)

**Creating a bot that can answer questions based on Wikipedia articles**

def qna\_article(topic, question):  
 chat = ChatOpenAI(max\_tokens = 500)  
 loader = WikipediaLoader(query=topic, load\_max\_docs = 1)  
 data = loader.load()  
 first\_record = data[0]  
 page\_content = first\_record.page\_content  
 title = first\_record.metadata['title']  
 summary = first\_record.metadata['summary']  
 user\_question = question  
  
 human\_template = "Read the paper with the title: '{title}'\n\n And Content: {content} and answer the questions {user\_question} related to the article"  
  
 human\_message\_prompt = HumanMessagePromptTemplate.from\_template(human\_template)  
  
 chat\_prompt = ChatPromptTemplate.from\_messages([human\_message\_prompt])  
 prompt = chat\_prompt.format\_prompt(title = title, content = summary, user\_question = question)  
  
 response = chat(messages = prompt.to\_messages())  
  
 return response.content

qna\_article('India', 'How many languages are being spoken in India?')

'India is a multilingual country with a diverse linguistic landscape. There are 22 officially recognized languages in India, as listed in the Eighth Schedule of the Indian Constitution. In addition to these official languages, there are hundreds of other languages spoken by different communities across the country.'

That's all about document loaders.