LangChain Document Loaders

Detailed Notes on Document Loading for RAG Systems

Introduction

In a Retrieval-Augmented Generation (RAG) pipeline, **Document Loaders** are the first stage responsible for importing unstructured data (text, PDFs, websites, etc.) into LangChain's processing ecosystem.

The loaded documents are then split, embedded, and stored in vector databases for semantic retrieval.

Definition

Document Loaders are utility components that read data from various file types or online sources and return them as standardized `Document` objects with `page_content` and `metadata`.

Role of Document Loaders in RAG

- Serve as the **data ingestion layer** in a RAG system.
- Convert raw data into a uniform document structure.
- Support multiple data sources — local files, APIs, web pages, cloud drives, etc.
- Output is used by **Text Splitters** and **Vector Stores** in the next steps.

Pipeline:

`Document Loader → Text Splitter → Embedding Model → Vector Store → Retriever → LLM`

Commonly Used Document Loaders in LangChain

LangChain provides a large collection of built-in loaders, each designed to handle a specific data format or source. Below are the most frequently used ones in real-world RAG systems.

1. `PyPDFLoader`

Purpose: Load text content from PDF files page by page.
Module: `from langchain_document_loaders import PyPDFLoader`

Example

```
from langchain_community.document_loaders import PyPDFLoader

loader = PyPDFLoader("sample.pdf")
documents = loader.load()

print(documents[0].page_content)
```

Notes:

- Extracts text from each PDF page separately.
- Metadata includes page number and file path.
- Works well for academic papers and reports.

2. `TextLoader`

Purpose: Load plain text files (`.txt`) into LangChain.
Module: `from langchain_community.document_loaders import TextLoader`

Example

```
from langchain_community.document_loaders import TextLoader

loader = TextLoader("notes.txt")
docs = loader.load()
print(docs[0].page_content)
```

Notes:

- Simplest loader, ideal for local text files.
- Metadata contains file path only.

3. `DirectoryLoader`

Purpose: Load all files from a directory recursively.
Module: `from langchain_community.document_loaders import DirectoryLoader`

Example

```
from langchain_community.document_loaders import DirectoryLoader

loader = DirectoryLoader("data/", glob="**/*.txt")
documents = loader.load()
print(len(documents))
```

Notes:

- Automatically detects file extensions.
- Internally uses the appropriate loader (e.g., TextLoader or PyPDFLoader).
- Useful for bulk data ingestion.

4. `WebBaseLoader`

Purpose: Load content directly from web URLs.
Module: `from langchain_community.document_loaders import WebBaseLoader`

Example

```
from langchain_community.document_loaders import WebBaseLoader

loader = WebBaseLoader("https://en.wikipedia.org/wiki/LangChain")
docs = loader.load()
print(docs[0].metadata)
```

Notes:

- Fetches HTML, extracts main text via BeautifulSoup.
- Preserves metadata such as URL and title.

5. `UnstructuredFileLoader`

Purpose: Handle mixed or unknown file types using the `unstructured` library.
Module: `from langchain_community.document_loaders import UnstructuredFileLoader`

Example

```
from langchain_community.document_loaders import UnstructuredFileLoader

loader = UnstructuredFileLoader("report.docx")
docs = loader.load()
print(docs[0].page_content)
```

Notes:

- Supports PDFs, DOCX, HTML, PPTX, and more.
- Uses heuristics to extract readable text blocks.

6. `CSVLoader`

Purpose: Load CSV data where each row is treated as a separate document.
Module: `from langchain_community.document_loaders import CSVLoader`

Example

```
from langchain_community.document_loaders import CSVLoader

loader = CSVLoader(file_path="data.csv")
docs = loader.load()
print(docs[0].metadata)
```

Notes:

- Each row becomes a LangChain Document.
- Metadata includes column names.

7. `YoutubeLoader`

Purpose: Load transcribed YouTube video content.
Module: `from langchain_community.document_loaders import YoutubeLoader`

Example

```
from langchain_community.document_loaders import YoutubeLoader

loader = YoutubeLoader.from_youtube_url(
    "https://www.youtube.com/watch?v=dQw4w9WgXcQ",
    add_video_info=True
)
docs = loader.load()
```

Notes:

- Uses YouTube transcript API.
- Includes metadata such as video title, duration, and channel.

8. `Docx2txtLoader`

Purpose: Load Microsoft Word (`.docx`) files using `docx2txt`.
Module: `from langchain_community.document_loaders import Docx2txtLoader`

Example

```
from langchain_community.document_loaders import Docx2txtLoader

loader = Docx2txtLoader("resume.docx")
docs = loader.load()
print(docs[0].page_content)
```

Notes:

- Extracts readable text from DOCX files.
- Metadata includes file name and path.

Comparison Table

Loader	**File Type / Source**	**Main Use Case**
PyPDFLoader	PDF Documents	Academic papers, research reports
TextLoader	Plain text files	Notes, logs, articles
DirectoryLoader	Multiple files in folder	Bulk ingestion
WebBaseLoader	Web URLs	News sites, Wikipedia pages
UnstructuredFileLoader	Mixed formats	Flexible data sources
CSVLoader	CSV tables	Datasets, structured info
YoutubeLoader	YouTube transcripts	Educational / tutorial videos
Docx2txtLoader	Word documents	Reports, resumes, books

Best Practices

- Use **DirectoryLoader** for batch ingestion.
- For multi-format projects, use **UnstructuredFileLoader**.
- Clean and normalize text before splitting.
- Store `metadata` (source, page, URL) for better retrieval tracing.

Creating Custom Document Loaders

In real-world projects, data often resides in APIs, databases, cloud services, or custom formats that are not directly supported by LangChain's built-in loaders. In such cases, you can define a **Custom Document Loader** by subclassing the `BaseLoader` class.

Definition

A **Custom Document Loader** is a user-defined class that inherits from `BaseLoader` and implements the `load()` method to return a list of `Document` objects.

Structure of a Custom Loader

Every custom loader must:

1. Inherit from `langchain.document_loaders.base.BaseLoader`

2. Implement the `load()` method
3. Return a list of `Document` objects with both content and metadata

Custom Loader Template

```
from langchain_community.document_loaders.base import BaseLoader
from langchain_core.documents import Document

class MyCustomLoader(BaseLoader):
    def __init__(self, source):
        self.source = source

    def load(self):
        # 1. Read or fetch the data
        raw_data = self._fetch_data(self.source)

        # 2. Convert it into a Document
        docs = [Document(page_content=raw_data,
                         metadata={"source": self.source})]
        return docs

    def _fetch_data(self, source):
        # Implement custom data fetching logic
        return "Sample text loaded from source!"
```

Example: API-Based Document Loader

Suppose you want to fetch articles from a public REST API (e.g., News API or Medium feed). A simple custom loader might look like this:

Example: API Loader

```
import requests
from langchain_community.document_loaders.base import BaseLoader
from langchain_core.documents import Document

class NewsAPILoader(BaseLoader):
    def __init__(self, api_url):
        self.api_url = api_url

    def load(self):
        response = requests.get(self.api_url)
        data = response.json()

        documents = []
        for article in data['articles']:
            content = article['title'] + "\n" + article['description']
            metadata = {"source": article['url'],
                        "author": article.get('author', 'unknown')}
            documents.append(Document(page_content=content,
                                      metadata=metadata))
        return documents

# Usage
loader = NewsAPILoader("https://newsapi.org/v2/top-headlines?
country=us&apiKey=YOUR_KEY")
docs = loader.load()
print(docs[0].page_content)
```

Key points:

- Fetches data dynamically from an API.
- Converts each record (article) into a separate Document.
- Includes useful metadata for traceability.

Example: Database Document Loader

You can also connect to SQL or NoSQL databases to load documents.

Example: Database Loader

```
import sqlite3
from langchain_community.document_loaders.base import BaseLoader
from langchain_core.documents import Document

class SQLiteLoader(BaseLoader):
    def __init__(self, db_path, query):
        self.db_path = db_path
        self.query = query

    def load(self):
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()
        cursor.execute(self.query)
        rows = cursor.fetchall()
        conn.close()

        documents = [
            Document(page_content=row[1],
                     metadata={"id": row[0], "source": self.db_path})
            for row in rows
        ]
        return documents

# Usage
loader = SQLiteLoader("mydata.db", "SELECT id, text FROM articles")
docs = loader.load()
print(docs[0].metadata)
```

Example: Google Drive Loader (Conceptual)

If your organization stores data in Google Drive, you can use Google API to load them as Documents.

Example: Google Drive Loader (Conceptual)

```
from googleapiclient.discovery import build
from langchain_community.document_loaders.base import BaseLoader
from langchain_core.documents import Document

class GoogleDriveLoader(BaseLoader):
    def __init__(self, folder_id, service):
        self.folder_id = folder_id
        self.service = service

    def load(self):
        results = self.service.files().list(
            q=f"'{self.folder_id}' in parents"
        ).execute()

        documents = []
        for file in results.get('files', []):
            file_id = file.get('id')
            name = file.get('name')
            content = self._read_file(file_id)
            documents.append(Document(
                page_content=content,
                metadata={"name": name, "id": file_id}
            ))
        return documents

    def _read_file(self, file_id):
        # Download or read file contents from Google Drive
        return "Sample Google Drive file content"
```

Best Practices for Custom Loaders

- Always include clear and descriptive metadata (e.g., source URL, database ID, timestamp).
- Handle exceptions and missing data gracefully.
- Keep `load()` methods lightweight — offload complex logic to helper functions.
- Cache or paginate data when loading from APIs or large databases.

- Consider subclassing existing loaders if the data format is similar.

When to Build Custom Loaders

- Data comes from APIs or internal services (not local files).
- The source has a unique format or access mechanism.
- You need specialized metadata fields.
- Integration with enterprise data systems (CRM, ERP, etc.).

Summary

Document Loaders are crucial for importing and structuring external knowledge into LangChain's ecosystem. While built-in loaders handle most file formats, **Custom Loaders** allow seamless integration of proprietary or dynamic data sources like APIs, databases, and cloud drives. Together, they form the backbone of any scalable, data-driven RAG system.

Conclusion

Document Loaders form the foundation of a RAG pipeline by transforming diverse data sources into standardized LangChain Document objects. Choosing the right loader ensures accurate, efficient, and scalable document ingestion for downstream tasks like chunking, embedding, and retrieval.