Section 3: Data Ingestion and Data Parsing Techniques
Course: Ultimate RAG Bootcamp Using LangChain, LangGraph and LangSmith
Section Number: 3
Total Videos: 9
Date Created: 2024

Video 1: Document Structure in LangChain
Video Order: 1/9
Topics: LangChain document structure, Page content vs. metadata, Document loaders, Text splitters, Project setup notes, Why metadata matters
Difficulty: Beginner

Content
Hello guys.

So we are going to continue the discussion with respect to **data ingestion**. Already in our previous video, we created our **virtual environment** and set up the **project structure**. Now, in this video and in the upcoming series of videos, we are going to focus on **data ingestion and parsing**.

Since we are using **LangChain** for data ingestion, the data will arrive in **different source formats**: PDFs, text files, Word documents, and more. I will try to show you multiple techniques to read these files with the **recent LangChain version 0.3**.

One very important point: as soon as we read documents, **LangChain expects us to convert that data into a *Document* structure**. I've included a diagram in the resources section titled **"LangChain Document Structure"**—this is a very important concept. For every document we read and plan to store in a vector database, we include key components like **page_content** and **metadata** (the metadata is a dictionary). This is the structure we will follow: read the data → convert it to the **Document** structure → (later) embed and store in a vector store. This step also lets us **enrich** content with additional metadata.

Here's the idea:

- **page_content**: a string containing the main text we want to embed and search.
- **metadata**: a dictionary with additional information (source, author, page, date, etc.) that helps retrieval and filtering.

There are many **document loader** techniques available in LangChain. We will focus on both **custom** and **in-built** techniques because parsing is a very important skill.

Let's start.

Introduction to Data Ingestion

Data ingestion means reading different kinds of data. For this, we'll import some libraries. If a library like pandas is missing, install it (for example: uv add pandas or your chosen package manager). As we add libraries, your pyproject.toml (or requirements) will update. If any code doesn't work, double-check the **installed versions** vs. the versions I use—mismatches can cause breaking changes over time.

Next, we'll import the basic types (List, Dict, Any) and pandas, and then the **Document** structure and **text splitters** from LangChain 0.3. For splitters, we will look at:

- **RecursiveCharacterTextSplitter**
- **CharacterTextSplitter**
- **TokenTextSplitter**

We'll use each and understand the differences as we go ahead. After setting this up, we'll just print a small message like "setup completed" so you know the environment is ready.

Understanding the LangChain Document Structure

This structure will play a **very important role** when you start building RAG pipelines. Initially, whenever you read any data, you need to convert it into **Document** objects. From the diagram, there are two key parts:

- **page_content** → the main text
- **metadata** → additional information about that text

Let's **create a simple Document** conceptually (code will come later):

- page_content: "This is the main text content that will be embedded and searched."
- metadata: a dictionary like:
    - source: "example.txt"
    - page_number: 1

    - author: "Krish"
    - date_created: "2024-01-01"
    - (any other relevant fields you want)

If you print the document, you'll see both parts:

- Access content via doc.page_content
- Access metadata via doc.metadata

Why metadata is crucial:

- **Filtering & search**: restrict results to a source, author, section, page range, etc.
- **Tracking sources**: show the user *where* the answer came from.
- **Providing context in responses**: include citation details or section names.
- **Debugging & auditing**: trace how a result was produced.

When we store content **with metadata** in our vector database, we can answer additional questions like "Who is the author?" because that information travels with each chunk. For advanced RAG, metadata becomes even more powerful.

From Loaders to Documents

When we use **LangChain document loaders** (for PDFs, text, Word docs, websites, etc.), the **return type is Document (or a list of Documents)**. That means, after loading, you already have objects with **page_content** and **metadata**. You can inspect their types and fields directly, and you're ready for **chunking** and **embedding** next.

Remember the ingestion diagram: after loading a document, we typically pass it to a **document/text splitter** to break it into **chunks**. We will use different splitters and compare them in practice.

Video 2: Ingesting and Parsing Text Data Using Document Loaders
Video Order: 2/9
Topics: Text files, TextLoader, DirectoryLoader, Document structure (page_content & metadata), UTF-8 encoding, glob patterns, pros/cons
Difficulty: Beginner

Content
Hello guys.

So we are going to continue the discussion with respect to RAG. Already in our previous video, we have understood the entire document structure inside data ingestion. So first of all, one type of data source files that we are going to read is a **text file**.

What we are going to do here is the simplest case: reading **.txt** files. You can refer to the earlier diagram for the **document data structure**—how the page_content is created, how metadata is created. We also wrote some code and understood that there are two important fields in each **Document**:

- page_content
- metadata

Document is a structure provided by LangChain. Anything that we read from supported data sources is converted into one or more **Document** objects, each with a page_content and metadata.

Let me now show you how to read text files. This is the simplest case you'll see.

Creating Sample Text Files (Setup)

Imagine some of your data is present inside .txt files. First, we'll create a simple directory and sample text files.

- Import os.
- Use os.makedirs() to create a directory inside the project where we're doing data parsing.
- Create a data/ folder, and inside it a text_files/ folder.
- Use exist_ok=True so it won't error if the folder already exists.

After creating the folders, we'll create **sample text files** using a small dictionary of {file_path: content} pairs, for example:

- data/text_files/python_intro.txt
- data/text_files/machine_learning_basics.txt

We'll iterate over the dictionary and, for each (file_path, content) pair:

- open(file_path, "w", encoding="utf-8") and write the content.
- Print a message like "Sample text file created."

At the end, you should have two files in data/text_files/:

- python_intro.txt
- machine_learning_basics.txt

Reading a Single Text File with TextLoader

The first method is reading a **single** file using **TextLoader**.

- Import TextLoader (commonly from langchain_community.document_loaders import TextLoader).
- Create a loader: loader = TextLoader("data/text_files/python_intro.txt", encoding="utf-8").
- Call documents = loader.load().

As discussed, the **return type** is a **list of Document** objects. If you print type(documents), you'll see it's a list. Each item looks like:

- Document(page_content=..., metadata={"source": "data/text_files/python_intro.txt"})

So even with the built-in loader, you get initial metadata automatically (e.g., source path). You can inspect the contents like this:

- Number of documents loaded
- Preview first 100 characters: documents[0].page_content[:100]
- Show metadata: documents[0].metadata

This confirms that a .txt file read via TextLoader yields a **list of Documents**, each with **page_content** and **metadata**.

Reading All Text Files in a Directory with DirectoryLoader

Now let's read **multiple** files directly from a directory using **DirectoryLoader**. This is suitable when you have many text files to ingest.

- Import DirectoryLoader from langchain_community.document_loaders.
- Create the loader:
    - directory_path = "data/text_files"
    - Use a **glob pattern** to match files, e.g., glob="**/*.txt" (recursive pattern: any subfolder, any .txt file).
    - Set loader_cls=TextLoader to specify the loader for each matched file.
    - Use loader_kwargs={"encoding": "utf-8"} so each text file is read with UTF-8.
    - Optionally show_progress=True to visualize the loading progress.
- Call documents = directory_loader.load().

Then, iterate over the results and print details:

- len(documents)
- For each doc in documents:
    - doc.metadata.get("source")
    - len(doc.page_content) (number of characters in the content)

You'll see output similar to:

- Document 1 → source=data/text_files/python_intro.txt, length=<chars>
- Document 2 → source=data/text_files/machine_learning_basics.txt, length=<chars>

DirectoryLoader: Characteristics, Advantages, Disadvantages

Advantages

- Load **multiple files** at once.
- Supports **glob patterns** for flexible matching (**/*.txt).
- Progress tracking with show_progress=True.
- **Recursive** directory scanning.

Disadvantages

- Typically assume files matching the glob are of the **same type** (e.g., all .txt) for a single loader class.
- Limited per-file error handling.
- Can be **memory intensive** for very large directories (many or large text files) since it scans and loads many files.

Summary of What We Did

- Created sample **text files** programmatically.
- Read a **single** text file via TextLoader → got a **list[Document]** with page_content and metadata.
- Read **multiple** text files via DirectoryLoader using a glob pattern and TextLoader as loader_cls.
- Saw how metadata (like source) is populated automatically and why it's helpful later in retrieval and filtering.
- Discussed the pros/cons of DirectoryLoader.

Video 3: Text Splitting Techniques
Video Order: 3/9
Topics: Text splitters, CharacterTextSplitter, RecursiveCharacterTextSplitter, TokenTextSplitter, Chunk size, Chunk overlap, Separators, Context window limits
Difficulty: Beginner

Content
Hello guys.

So we are going to continue the discussion with respect to our **data ingestion pipeline**. Already in our previous video, we have seen how to read a .txt file and convert it into a **Document** using the document loaders available in LangChain.

Now, let me go back to the RAG architecture diagram we saw earlier. In the **data ingestion phase**, we may have different files: PDFs, web pages, text files, etc. After reading a file, the return type is a **Document** (or list of Documents). Then we apply something called a **document splitter** (also called a **text splitter**). The main goal is to **split the document into chunks**.

In this video, I'm going to discuss some of the **text splitting strategies** available in LangChain. As we go ahead, we'll also look at more techniques when we move toward **advanced RAG**.

The idea of text splitting is simple: we split the loaded documents into **smaller chunks** because **LLMs have context window limitations**. Smaller, coherent chunks help retrieval and improve downstream answer quality.

Importing Text Splitters

We will import the splitters from langchain.text_splitter (or the updated module path as per LangChain 0.3):

- **CharacterTextSplitter**
- **RecursiveCharacterTextSplitter**
- **TokenTextSplitter**

We will print our loaded documents (list of Document). Then we'll take one Document (e.g., documents[0]) and access doc.page_content to apply the different splitters.

Method 1: CharacterTextSplitter

The **CharacterTextSplitter** performs splitting based on **characters** using a specified **separator**.

- **Separator**: for example, a newline ("\n").
- **Chunk size**: e.g., 200 characters (max characters per chunk).
- **Chunk overlap**: e.g., 20 characters (the last 20 chars of one chunk repeat at the start of the next).
- **Length function**: standard len.

We create the splitter and call either split_text(text) (if we're splitting a string) or split_documents([doc]) (if we want Document-aware splitting). Here we'll illustrate split_text(text).

- Print len(char_chunks) to see how many chunks were created.
- Print a few chunks (e.g., char_chunks[0], char_chunks[1]) to inspect content.

If the **separator** is a newline and our text contains natural line breaks, you may observe **clean splits** with **little to no visible overlap** in the printed chunks (even if chunk_overlap is set), because the splitter respects the separator boundaries.

Method 2: RecursiveCharacterTextSplitter (Recommended)

The **RecursiveCharacterTextSplitter** is the **most recommended** general-purpose splitter. It tries multiple **separators** in order (e.g., "\n\n", "\n", space, etc.) and recursively backs off to smaller units when larger separators don't fit the chunk size.

- Provide a list of **separators** (e.g., ["\n\n", "\n", " ", ""]).
- Set chunk_size=200, chunk_overlap=20, length_function=len.

Call split_text(text) and inspect results:

- You may see a different **number of chunks** (e.g., 6 instead of 4) compared to the simple character splitter, because the recursive method respects document structure and splits more intelligently.
- If your text has many clean breakpoints (headings, blank lines), you may not **see** the overlap in the printed output, even though chunk_overlap is set; that's just because the splits landed on clean boundaries.

To **demonstrate overlap clearly**, use text **without natural breakpoints** and restrict the separator list (e.g., only a space separator), with smaller chunk_size like 80 and chunk_overlap=20. Then print consecutive chunks to see repeated phrases (e.g., "...and it is also" appearing at the end of one chunk and the beginning of the next).

Method 3: TokenTextSplitter

The **TokenTextSplitter** splits by **tokens** (not just raw characters). Depending on the tokenizer, spaces and punctuation are accounted for in token counts.

- Example settings: chunk_size=50 tokens, chunk_overlap=10 tokens.
- Call split_text(text) and inspect the produced chunks.

This method is often slower than character-based splitters but is useful when working with **token-limited models** (you want chunks measured in tokens, not characters).

Making Overlap Visible (Examples)

If you didn't see overlap earlier, there are a few reasons:

- **Chunk size** and **overlap** may be relatively small.
- Your separators caused **clean splits** (e.g., splitting at headings/blank lines).

To force visible overlap:

- Reduce chunk_size.
- Increase chunk_overlap.
- Limit separators (e.g., only spaces) so splits are more likely to cross mid-sentence.

Then print adjacent chunks to see repeated phrases across boundaries:

- Example: chunk end "...is even longer than" and the next chunk starts with "...is even longer than...".

When to Use Which Splitter

CharacterTextSplitter

- **Pros**: Simple, predictable; good for structured text with clear delimiters.

- **Cons**: May break mid-sentence if separators are not aligned.
- **Use when**: Your text has clear, consistent delimiters and you want fast, straightforward splitting.

RecursiveCharacterTextSplitter (default choice)

- **Pros**: Respects text structure; tries multiple separators; best general-purpose splitter.
- **Cons**: Slightly more complex; may take more time than simple character splits.
- **Use when**: Default for most text; you want structure-aware splitting and good chunk quality.

TokenTextSplitter

- **Pros**: Measures chunks by tokens; aligns with LLM token limits.
- **Cons**: Slower; depends on tokenizer behavior.
- **Use when**: Working with strict **token** constraints or models where token budgeting is critical.

Why Chunking Matters (Retrieval Perspective)

At the end of ingestion, these chunks are **embedded** and stored in a **vector database**. During retrieval, a query may match content spread across multiple overlapping chunks. Overlap increases the chance that all relevant context is captured by the top-k retrieved chunks. This leads to better grounding and answer quality in the generation phase.

Video 4: Ingestion and Parsing PDF Documents
Video Order: 4/9
Topics: PDF ingestion, PyPDFLoader, PyMuPDFLoader, UnstructuredPDFLoader (intro), Documents (page_content & metadata), Page-level info, Loader comparison
Difficulty: Beginner

Content
Hello guys!

Super excited that we have now completed at least one specific part wherein we are able to read from a text document or text file, and then we are also able to split them into chunks. Right now it's time to go ahead and show you how we can **load PDF files**, because this is one of the most common use cases. Many companies have a huge amount of PDFs, so it is necessary that you know how to read a PDF with LangChain.

Inside my data/ folder, I have uploaded a pdf/ folder, and there is a file named **attention.pdf**. This specific PDF is a **research paper**. We will read this PDF, convert it into **Document** objects, and then see how we can apply splitting techniques over the extracted text.

First, I'll create another notebook/file for this part, e.g., data_parsing_pdf.ipynb, because PDF handling is a very common real-world requirement.

Loading PDFs with LangChain (Approaches)

We will import loaders from langchain_community.document_loaders and demonstrate three approaches (we will implement two now and introduce the third for later):

1. **PyPDFLoader** (often recommended, simple and reliable)
2. **PyMuPDFLoader** (fast, strong text extraction; supports images)
3. **UnstructuredPDFLoader** (we'll cover later, useful for messy PDFs)

Method 1: PyPDFLoader

PyPDFLoader is a common and recommended way to read PDFs.

- Create a loader with the file path, e.g., data/pdf/attention.pdf.
- Call .load() to get a **list of Document** objects, one per page (typically).

- Each Document contains:
    - **page_content** → extracted text for that page
    - **metadata** → fields like page, source, creator, producer, author, timestamps, etc.

After loading, print diagnostics:

- **Number of pages** (e.g., 15 pages loaded)
- **Page 1 preview** → first 100 characters
- **Metadata** → show how much information is automatically captured

This is an easy way to read a PDF and immediately get page-oriented Documents you can later split and embed.

Method 2: PyMuPDFLoader (fitz)

PyMuPDFLoader uses the pymupdf (fitz) engine under the hood.

- You may need to install pymupdf first.
- Create the loader with the same file, call .load().
- Inspect the returned Documents and metadata.

Why consider PyMuPDFLoader?

- Generally **fast**
- Often **robust** text extraction
- Supports **image extraction** and richer PDF features

If you see import errors, install the dependency (e.g., pymupdf) and re-run.

(Later) Method 3: UnstructuredPDFLoader

We will discuss **UnstructuredPDFLoader** in a later video when we dive into **cleaning and advanced PDF processing**. It is helpful for:

- Complex layouts
- PDFs with images, tables, headers/footers

- Needing heuristic segmentation before chunking

Quick Look at the PDF Example

The example PDF is the classic research paper **"Attention Is All You Need."** Using the loaders above, we obtain page-wise Documents. In the **metadata**, you'll typically see fields like creator, producer, modDate, path, and the current **page** number. In the **page_content**, you'll see the extracted text. This structure prepares the content for **splitting** and **embedding**.

Loader Comparison (Cheat Sheet)

PyPDFLoader

- **Pros**: Simple, reliable; good for most PDFs; preserves page numbers; basic text extraction
- **Cons**: Can struggle with image-heavy or weirdly encoded PDFs
- **Use when**: You want a straightforward loader that works well on standard PDFs

PyMuPDFLoader

- **Pros**: Fast; strong text extraction; **image extraction support**
- **Cons**: Requires pymupdf dependency; sometimes different extraction quirks
- **Use when**: Speed matters or you need better extraction on tricky PDFs

UnstructuredPDFLoader (preview)

- **Pros**: Powerful heuristic parsing for messy layouts
- **Cons**: Heavier dependency stack; more configuration
- **Use when**: You need advanced, layout-aware preprocessing before chunking

---

Video 5: Handling Common PDF Issues
Video Order: 5/9
Topics: PDF parsing challenges, whitespace & newline cleanup, ligature fixes, OCR mention, metadata enrichment, SmartPDFProcessor class, recursive splitting, empty-page handling, robust ingestion pipeline
Difficulty: Beginner → Intermediate

Content
Hello guys.

So we are going to continue our discussion with respect to **data parsing for PDFs**. Already in our previous video, we discussed two important libraries for reading PDFs—**PyPDFLoader** and **PyMuPDFLoader**—and we saw how to read a PDF file, extract details, and inspect some of the **metadata**.

In this video, we're going to discuss **common PDF challenges** that you need to handle. The purpose is simple: if your PDF has plain text, parsing is easy. But PDFs often store text in complex ways (tables, columns, scanned images), include formatting artifacts (excess whitespace, broken lines, hyphenation), or contain **special/encoded characters** (like ligatures). We need to **clean** these issues so the downstream chunking, embedding, and retrieval are effective.

Why PDFs Are Tricky

- **Tables & multi-column layouts** can scramble text order.
- **Scanned PDFs** require **OCR** to extract text.
- **Extraction artifacts**: stray newlines (\n), page headers/footers, hyphenations, broken words.
- **Special characters/ligatures**: fi, fl, etc., due to font encodings.
- **Whitespace noise**: multiple spaces or irregular spacing.

We'll look at a simple **cleaning function** to normalize raw text after extraction.

Example: Raw vs Cleaned Text

Suppose a raw page contains something like:

"Company Financial Report\n\n\nThe financial performance for fiscal year 2024 shows significant growth in profitability. Revenue increased by 25%. ..."
with lots of extra spaces/newlines and encoded ligatures.

A **cleaning function** can:

- Split on whitespace and **rejoin with single spaces** (normalizing spacing).
- Replace **ligatures** like fi → fi, fl → fl.
- (Later) Add more rules for hyphenation, headers/footers, etc.

After cleaning, the text becomes a single, readable paragraph with correct characters, ready for splitting and embedding.

Building a Reusable Processor (Class Design)

To make this robust, we'll create a class called **SmartPDFProcessor** that handles PDF loading, cleaning, **smart chunking**, and **metadata enrichment**.

Constructor (__init__)

- Parameters: chunk_size=1000, chunk_overlap=100.
- Initializes a **RecursiveCharacterTextSplitter** with:
    - chunk_size
    - chunk_overlap
    - separators=[" "] (split primarily on spaces for consistent mid-sentence handling)

Private method: _clean_text(text: str) -> str

- Normalizes whitespace (e.g., ' '.join(text.split())).
- Replaces common ligatures/encoded characters (e.g., fi→fi, fl→fl).
- Returns cleaned text. (Extend later for OCR/table handling.)

Public method: process_pdf(pdf_path: str) -> List[Document]

1. **Load** the PDF via PyPDFLoader(pdf_path).load() to get page-wise Documents.
2. Iterate pages with enumerate(pages).

3. **Clean** each page's page_content with _clean_text.
4. **Skip empty pages** (e.g., if len(clean_text.strip()) < 50).
5. **Create chunks** with self.text_splitter.create_documents([clean_text], metadata=...).
    - Enrich each chunk's metadata using the original page.metadata plus:
        - page_number
        - total_pages
        - chunk_method (e.g., "recursive_space")
        - char_count
6. Append all per-page chunks to a processed_chunks list and return it.

This design ensures that **every chunk** in your vector database carries **useful metadata** for filtering, tracing sources, and improving retrieval quality.

Running the Processor

- Instantiate: processor = SmartPDFProcessor().
- Call: smart_chunks = processor.process_pdf("data/pdf/attention.pdf").
- Expect a message like: **Processed into 49 smart chunks** (depends on your chunking settings and the PDF content).
- Inspect metadata for a few chunks to confirm fields like page_number, total_pages, chunk_method, source, etc., are present.

Why Metadata Matters

When these chunks are embedded and stored in your vector DB, metadata allows you to:

- **Filter** by page_number, section, or source.
- Provide **citations** (page and document).
- Improve **retrieval precision** (e.g., restrict to particular sections/pages in follow-up queries).
- **Debug/audit** the pipeline when results look off.

Video 6: Ingestion and Data Parsing Word Documents
Video Order: 6/9
Topics: Word documents, Docx2txtLoader, UnstructuredWordDocumentLoader, Elements mode, Metadata, Chunking strategy
Difficulty: Beginner → Intermediate

Content
Hello guys.

So we are going to continue the discussion with respect to **data parsing techniques**. Already in our previous video, we finished the task of **data parsing PDFs** and saw which common functionalities we can use to handle text data extracted from PDFs. We also showed how to create a class and how we'll keep implementing each functionality as we go ahead.

At the end of the day, we are still focusing on the main goal: **read data from a data source**, apply **splitting**, convert it into **chunks**, and while creating these chunks, **add metadata**. Later, we will see the advantages of this metadata during retrieval.

In this specific video, under **data ingestion**, we will create one more file (e.g., data_parsing_doc.ipynb) and focus on **Word documents**. Your problem statements may include data in .docx format—project proposals, reports, HR policies, etc. Here's an example: a **project proposal** for a RAG implementation. I've created a simple document with an **executive summary**, **job responsibilities**, and a few structured sections. We'll read this Word doc, extract content, add metadata, and (optionally) chunk it.

Let's go step by step.

Word Document Processing Overview

For Word document processing, we'll use two important loaders from langchain_community.document_loaders:

1. **Docx2txtLoader** — quick text extraction from .docx
2. **UnstructuredWordDocumentLoader** — structure-aware parsing using the **elements** pipeline

You may also see Python libraries like python-docx and docx2txt in the environment for working with .docx directly, but here we'll focus on the **LangChain loaders** so the output is already in **Document** form (page_content + metadata).

Method 1: Docx2txtLoader (Simple Text Extraction)

- Create a Word doc in data/word_files/ named proposal.docx (e.g., a RAG implementation proposal with objectives, scope, milestones).
- Use Docx2txtLoader to read it:
    - loader = Docx2txtLoader("data/word_files/proposal.docx")
    - docs = loader.load() → returns a **list[Document]**
- Print diagnostics:
    - len(docs) → how many Document objects were returned (often 1 for simple docs)
    - docs[0].page_content[:100] → preview content
    - docs[0].metadata → see fields like source (file path), etc.

If everything is fine, you should see content like: Executive Summary ... and metadata showing the file path. This confirms the loader converted the Word file into a **Document** with **content and metadata**.

Method 2: UnstructuredWordDocumentLoader (Elements Mode)

For more complex .docx files with headings, lists, tables, and mixed formatting, use **UnstructuredWordDocumentLoader**.

- Initialize with mode="elements" to preserve structural elements:
    - un_loader = UnstructuredWordDocumentLoader("data/word_files/proposal.docx", mode="elements")
    - un_docs = un_loader.load()
- This may take longer on first run (heavier processing). The result is a **list[Document]** where each item may correspond to an **element**:
    - Example elements: Title, NarrativeText, ListItem, Table, etc.
- Print diagnostics:

- len(un_docs)
- For the first few elements, print doc.metadata and doc.page_content

You might see output like “Loaded 20 elements” with metadata including source, file_directory, file_name, last_modified, languages, category, etc. The **category** reflects the element type (Title, NarrativeText, etc.). Tables are often parsed as separate elements—very handy for downstream processing.

What About Chunking?

When mode="elements" is used, the loader already **segments** the document into meaningful pieces. You can:

- **Use elements as-is** as chunks (each element is small and coherent), or
- Further apply a **text splitter** (e.g., RecursiveCharacterTextSplitter) to long elements (e.g., long NarrativeText) with settings like chunk_size=500, chunk_overlap=50, and inject metadata such as section, category, element_index, source, etc.

This two-level approach (elements → splitter) gives excellent control and often better retrieval quality.

Example Workflow (Putting It Together)

1. **Load** the .docx using one of the loaders above.
2. **Inspect** the returned Documents → check page_content and metadata.
3. **Normalize/Clean** the text if needed (whitespace, ligatures rare in .docx but can normalize newlines).
4. **Chunk**:
    - If using Docx2txtLoader, run a splitter over the full text.
    - If using UnstructuredWordDocumentLoader with mode="elements", treat each element as a base chunk and optionally split further.
5. **Enrich Metadata** for each chunk (e.g., source, category, heading, element_id, char_count).
6. **Store** the chunks in your vector database.

Notes on Performance & Trade-offs

- **Docx2txtLoader**
    - **Pros**: Fast, simple, minimal dependencies
    - **Cons**: Loses structural context (headings, lists, tables)
    - **Use when**: You just need text quickly from clean .docx
- **UnstructuredWordDocumentLoader (elements)**
    - **Pros**: Preserves structure; identifies element categories; captures tables as separate elements
    - **Cons**: Slower; heavier dependencies
    - **Use when**: You care about document **structure**, need **table** extraction, or want element-aware chunking

---

Metadata Matters (again!)

As with PDFs, rich metadata on Word-derived chunks improves retrieval and explainability:

- Filter by category (Title vs NarrativeText vs Table)
- Track source and file_name for citations
- Use heading or inferred section titles for **section-aware** retrieval

---

Wrap-up

In this video, we learned how to ingest **Word documents** using **Docx2txtLoader** and **UnstructuredWordDocumentLoader**. We saw how **elements mode** can yield multiple Document elements with structural metadata, which is great for high-quality chunking. From here, you can apply splitters as needed, enrich metadata, embed, and store in your vector DB.

In the **next video**, we'll discuss **CSV and Excel** files—how to read them and prepare the content for splitting and embeddings.

Thank you. Take care.

Section 3: Data Ingestion and Data Parsing Techniques
Course: Ultimate RAG Bootcamp Using LangChain, LangGraph and LangSmith
Section Number: 3
Total Videos: TBD
Date Created: 2024

Video 7: Parsing CSV and Excel Files
Video Order: 7/9
Topics: Structured data, CSVLoader, UnstructuredCSVLoader, Pandas, Excel processing, UnstructuredExcelLoader, Custom CSV processing, Metadata enrichment
Difficulty: Beginner → Intermediate

Content
Hello guys.

So we are going to continue the discussion with respect to **data parsing and data ingestion**. Already in our previous video, we have seen how to parse **Word** docs. And whenever we talk about all the previous parsing techniques that we have learnt, it is mostly for **unstructured data**, right? So in this particular video, we are going to go ahead and see how we can **ingest and parse structured data** like **CSV and Excel files**.

Okay.

So with respect to this, the first thing is that I will just go ahead and quickly create one more file. So let's say here I will go ahead and write **csv_excel_parsing.ipynb**. And this will basically be my fourth file. Okay I will go ahead and select the kernel quickly. I will create a markdown and let me just go ahead and write **CSV and Excel files**—right—and this is nothing but my **structured data**.

See, the main aim of creating this is that I really want to show you **each and every technique** of parsing from different data sources. Okay. So that tomorrow, if you're working on anything—right—if there is a CSV file, if there is some kind of Excel file, you should know what you're actually doing or how to probably go ahead and **ingest that and do the normal parsing** and **convert that into a document structure**.

Okay.

So the first thing that I'm going to import is nothing but, since we are going to work with CSV or Excel file, I think **pandas** will definitely be required. So I will go ahead and **install pandas**. Okay. So right now pandas is not there. But let's see whether LangChain is also coming or not. Okay.

So this is the problem that I'm facing. And you may also face whenever you create a new file—right—at that point of time, the libraries will not get loaded and you will not be able to quickly get all the suggestion, like how we used to get it over here. Okay, so for this you just need to go ahead and **restart** it once again. Restart your kernel completely. Okay. And once you restart your kernel I think you should be able to see it again.

Okay.

So first of all what I will do, I will quickly go ahead and see whether in my requirements.txt I have pandas or not. So here pandas is not there. I will go ahead and import pandas. Pandas. So here what I will do I will quickly write uv add -r requirements.txt. Okay, so pandas has been imported already I see okay somewhere pandas may be there and now I will quickly go ahead and write import pandas. Okay, so let's see. This is executing perfectly fine.

Okay.

So with respect to this, if you know how to use pandas, I think that will be more than sufficient in order to start, right. One more thing. What I will do, I will just go ahead to my folder location where I'm writing the code. I will again open up VS Code. Okay, I'll open the same project in VS Code. There is a reason why I'm doing it and I'll just let you know in some time. Okay? Why I am actually doing that right? So first of all what I will do, I will just close this. I will just close—or let me just close it quickly from here, okay. And then I will just go ahead and open my VS Code.

Now you'll be able to see that here. I will go ahead and select my kernel. So all the kernels will be getting loaded quickly. Yeah perfect. So I have loaded my kernel. Now you can see that my pandas is basically coming. See whenever things are not getting loaded, just try to close your VS Code or try to restart your kernel. I restarted my kernel. That thing also did not work. So with respect to every video, you know, I definitely will tell you some or the other things over here. So don't get bored with respect to that, okay?

Then I'm going to go ahead and import the os because this kind of suggestions I will not be getting like—when I say suggestions, the highlighting will not be coming up. And if highlighting is not coming up like you'll not understand where you're making a spelling mistakes and all. And then later on in the debugging section you'll be doing all those things. Okay.

Now quickly what I will do, I will go ahead and make one directory. So since we are working with CSV and Excel file, what I will do first task is basically to **create these files**. Okay. Then I will show you how you can go ahead and read that file and do the parsing also. So first of all what I'm doing I'm creating a **structured_files** folder inside my data folder. So I will just

go ahead and execute it. So if you see over here inside my data there is a structured_files folder, okay.

Now inside this folder I will go ahead and create a **simple DataFrame**. Okay. So if you know pandas simple DataFrame: see product—this is my column name. These are my data that is present inside the column. Another column—these are my data; price—these are my data; and description—these are my data, right. And this I'm trying to **convert into a DataFrame**, and I'm **saving that DataFrame** in a file which is called as products.csv.

So if you know about at least some of the data pre-processing techniques, if you have used it, I think this is the most simple way of creating a CSV file. Okay. So I'll quickly go ahead with this because these are some basic things which you should know in Python. Okay.

Now inside my data folder, if I go ahead and see there is a structured_files folder and there is a CSV file, and this is how my CSV file looks like. So one CSV file we have actually created it. And now similarly I will go ahead and **create one Excel file** also. Okay.

Now for Excel file I will again take some data and probably print it in over here. So let's say this is one important functionality which is called as **ExcelWriter** inside pandas. So here is my inventory.xlsx as a writer. And I'm trying to convert this into Excel. I'm using a sheet_name="products" and index=False. So here is my **entire summary data** which I'm going to put inside this particular Excel file. And then I'm taking the summary data, converting into a DataFrame and finally converting into an Excel with this particular sheet name and writer and index=False where I don't take my separate column name. Right. Whatever column is specifically given over here that only it will be taken.

So this code actually helps you to create some kind of **Excel file** over here. So if you just go ahead and execute this. So here, **No module named openpyxl** is coming. So what I will do, I will just go ahead and write it: openpyxl. So this is a library that we are going to specifically use it. So I will open my command prompt, uv add -r requirements.txt. Okay. So here you can see openpyxl (and et-xmlfile) is got loaded. Right. So I will go to this and again execute this. Now I think it should work.

So at the end of the day you will be able to see that along with my CSV file there will also be an **XLSX file**—right—Excel file. So two important files I have created: products.csv and inventory.xlsx. Very simple. I've just used pandas. One function is pd.ExcelWriter where we have to just give the sheet name and I will be able to create it. Right. And these are my information that is present over here.

Now let's start working on **CSV processing**. Right. So here we are going to specifically go ahead and work on how, if we are reading a CSV file, how should we go ahead and read it? And how should we basically **convert that into a Document**? Okay.

Again, from LangChain there is an inbuilt function which is called as **CSVLoader**. So I will go ahead and import from langchain_community.document_loaders import CSVLoader. Okay. And then from langchain_community.document_loaders import UnstructuredCSVLoader. So these two specific libraries we are going to use in order to load a CSV file. Okay.

So first path is nothing but **CSVLoader**. And we will just try to see how to work with the **method one**. So in the method one you can see **CSVLoader (row-based Documents)**. I'm using the CSVLoader; I have to give my path for the CSV file. We have to use encoding as UTF-8 and here additional you can give **delimiter** and **quotechar**. So delimiter I'm actually giving , since it is CSV (comma-separated file). We can give this. Quote character is nothing but quotes ".

Then we are using csv_loader.load() and I'm printing all the information like what are my documents and all. If you want I can also go ahead and print all the CSV documents. Okay csv_docs. So let's go ahead and print this. So here you can see this is one Document metadata—all the information. page_content—you can see everything is available inside that page content. Right. Yeah one. Then again my another Document metadata automatically I'm able to read it. So **based on row by row**—I think it is row by row—yeah. See row=2, row=1. All this information is here.

So row=2 is one Document. Okay. So one Document. So if you see this there are total how many rows? There are five rows. One, two, three, four, five. Right. So (0, 1, 2, 3, 4). Right. And if you go ahead and see over here that many number of Documents you'll be able to find out. Right. And here you can see row=4 and all the information—"1080p with noise cancellation"—is there. Here you can see 1080p with noise cancellation: these are my information of the columns. Quickly you can see that once we are reading this, automatically **row by row it is converting into Documents**. Okay. And the first Document is this information. The metadata information is also here. And here you can see, since it is reading a CSV file, automatically this function is so good that it is adding a metadata as row=0.

See, at the end of the day, you can also go ahead and **create your own custom Document** with **additional metadata information**. So this is important that you should know how to read something and how you can go ahead and customize this. I can also do this customization. Right. At the end of the day, I'm getting all the format in the Document. I can go ahead and **add more metadata** if I want, which I've already shown you before also, right, with respect to different different things.

So this was one. The second technique that you can specifically use is one more **UnstructuredCSVLoader**. Okay. So UnstructuredCSVLoader—what we can actually do and how we can actually read it—that is what we are basically going to discuss now. Okay.

Now quickly, let's do one thing. Let's try to see that **before going to unstructured**, you may be thinking that, "Krish, can we add more additional metadata information and all?" So for that what I will do, I will create a **custom CSV processing**. Right. And let's see this. Okay.

So I'm going to copy and paste this code—see **Custom CSV Processing**. So for this I will be requiring List. So I will go ahead and write from typing import List. And then I will also go ahead and import my **Document**. See I don't need to by-heart everything like where exactly it is or not. Okay. But it is good to know all these things. But it's okay—with practice you will get to know from where you are importing all the things, right? So from typing import List and this one.

See here what we are trying to do is that we are creating our **custom CSV processing** for better control. So here we have created a function process_csv_intelligently(file_path) -> List[Document]. I'll just give the file path and this will return a **list of Documents**. Okay. So it is going to return a list of Documents where we are also going to do the chunking—each and everything—okay. Automatically the chunking, parsing along with the metadata information in each and everything.

So first of all I'm reading the CSV file path. I've created an empty documents. **Strategy 1 is "one Document per row with structured content."** So what I will do, I will iterate through every row. And for that we use df.iterrows(). Okay, now what we are basically going to do is that I'm providing some of the content information over here. Right? This is my content information. Now see next step is that I'm creating the **Document**. So for the Document I will be using this content. This content is coming from where? See it is coming from this information—right—row by row. All this specific information I'm going to put inside this content. So this becomes my page_content.

Then metadata. So metadata I have my file_path. If you see file path is here. Then row_index from this index. You can get row_product, row_category, row_price and product_info. The same thing this CSVLoader is actually doing. And here you can see that how easily we have actually written our own **custom function** for the CSV processing. And that is how you just go ahead and, you know, create your own information with respect to this write-up. When you want some more additional information, you can just go ahead and write it.

Okay.

Now I definitely don't want to stop it over here—like you can do it from your own. Right. So once I execute this, you will be able to see that this CSV processing is there. Now, if I just call this function and if I give my path—my path, what path it is? Let's see this path. So let's say I give this path my CSV path, okay, and I give it over here. Now what output I'm actually going to get? Okay. So I'll just go ahead and write return documents. Okay.

Now, if I go ahead and execute this so you can see over here all the information is over here. row=0, row=1, row=2, row=3, row=4—all the information with respect to this is there, okay.

So I hope you got an idea about how you can do the **custom CSV processing**. At the end of the day, this is what you really want to create and how you want to create it. As you understand multiple use cases, you'll get to know about it.

Okay, now there may also be one more scenario, where I also want to probably go ahead and **create a summary**, something like that. Right. That I will show you as we go ahead. But just to make sure to show you the two differences like **CSV processing strategies** that we have discussed till now—you know: row-based and whether you should go ahead with this custom one—I'll just write the differences so that you get to know about this. Okay.

So here—**CSV Processing Strategies**:

- **Row-based (CSVLoader)** → *Simple*: one row → one Document; good for record lookups; **loses table context**.
- **Intelligent/custom processing** → *Preserves relationships*, create summaries, rich metadata, better for **Q&A**. Because here you can add more context, more information. In short, when you're adding more context, here you can also say that, "Hey, does this CSV have some other relationship with some other CSV file also?" Right. So that context can be put up in this metadata information. So based on the use cases, you think whether you need to probably go ahead with using CSVLoader or **intelligent processing**.

Okay.

Now coming to the **Excel processing**—again, guys, when we say coding, we should not restrict ourself in one thing. I should definitely show you **multiple ways** how to do this.

So now let's go ahead with **Excel processing**. For Excel processing I will show you one technique. So here you can just go ahead and see this, okay. And you'll be able to understand it anyway.

Method 1: Using pandas for full control

- Create a function process_excel_with_pandas(file_path) -> List[Document].

- Read the Excel file with pd.ExcelFile(file_path).
- For sheet_name in excel.sheets (or excel.sheet_names), load each DataFrame.
- Build a sheet_content string representing the table (you can format as CSV/TSV or Markdown if you want).
- Create a **Document** with page_content=sheet_content and metadata including: source, sheet_name, n_rows, n_cols.
- Append all to a documents list and return.

Once I execute this quickly I can just go ahead and print things. Okay. So here I'm giving my excel_docs, len(excel_docs), each and everything. So here you can see **two sheets have been processed**. So if you go ahead and see my excel_docs this is what is the information that I have: inventory, product summary—each and every information with respect to this, okay. So it is just reading from the inventory file.

Method 2: UnstructuredExcelLoader

- You can also use UnstructuredExcelLoader whenever you want.
- Import: from langchain_community.document_loaders import UnstructuredExcelLoader.
- Create the loader with the file path; call .load().
- As with other unstructured loaders, this may take **time** on larger or complex workbooks.
- **Advantages**: handles complex Excel features, **preserves formatting info**.
- **Disadvantages**: requires the unstructured library; slower.

Video 8: JSON Files Parsing and Processing
Video Order: 8/9
Topics: JSON ingestion, JSONLoader with jq_schema, Custom JSON parsing, Document structure, Metadata enrichment, API JSON handling
Difficulty: Beginner → Intermediate

Content
Hello guys.

So in this specific video we are going to discuss about **JSON parsing and processing**. Let's say you have JSON data—how do we ingest it, parse it, process it, and finally **convert it into a Document structure**? There may be scenarios where you are communicating with APIs and the responses are in JSON. This becomes a very important use case, and many practical pipelines will involve JSON.

First, we'll **create some JSON files** to work with.

Creating Sample JSON Files

We'll start by importing json and os, and creating a directory data/json_files/ (using os.makedirs(..., exist_ok=True)).

File 1: company_data.json
A nested JSON with a structure like:

- company: "TechCorp"
- employees: a list of employee objects (each with id, name, role, skills, projects)
- projects: list of project summaries
- departments: list of department names

We'll json.dump(...) this json_data to company_data.json.

File 2: events.json
Another JSON containing a list of **event logs** with fields like timestamp, event, user_id, page, etc. We'll save this to events.json.

After this step, you should have:

- data/json_files/company_data.json
- data/json_files/events.json

Method 1: Using LangChain's JSONLoader (with jq_schema)

We'll import JSONLoader from langchain_community.document_loaders.

Sometimes you don't want the **entire** JSON—just a specific **subtree** (e.g., each employee). JSONLoader supports a **jq_schema** parameter: a **jq** expression to select parts of the JSON.

Example: Extract each **employee** object as a separate Document.

- Initialize:
  employee_loader = JSONLoader(file_path="data/json_files/company_data.json", jq_schema=".employees[]", text_content=False)
- Call: employee_docs = employee_loader.load()

Notes:

- jq_schema=".employees[]" traverses into employees and emits **each element**.
- text_content=False returns the **full JSON object** in page_content (serialized), not just plain text.
- You may need to install jq support (the Python package used by the loader); ensure it's in your environment.

Result:

- You'll see something like: "Loaded 2 employee documents."
- Each Document has:
  - page_content: the serialized JSON for **one employee**
  - metadata: includes source and other fields

This is perfect when you want one vector **per employee** for retrieval.

Method 2: Custom JSON Processing (Full Control)

While the loader is convenient, sometimes you need more control. Let's build a small custom function that reads the JSON and **emits Documents with rich metadata**.

Function sketch: process_company_json(file_path) -> List[Document]

1. with open(file_path) as f: data = json.load(f)

2. Iterate over data.get("employees", []).
3. For each employee, construct a **content string** (or keep JSON) that summarizes key fields: id, name, role, skills, and a list of projects.
4. Create a **Document**:
    - page_content: the formatted text (or json.dumps(emp, ensure_ascii=False))
    - metadata: { "source": file_path, "record_type": "employee", "employee_id": emp["id"], "name": emp["name"], "projects": [..], "skills_count": len(emp.get("skills", [])) }
5. Append to documents and return the list.

This approach lets you **normalize text**, **rename fields**, **filter sensitive keys**, and inject **domain-specific metadata** to improve retrieval, filtering, and analytics later.

Applying to Event Logs (Assignment-style Tip)

For events.json, follow a similar pattern:

- Iterate events, create one Document per event (or per user, grouped).
- **Metadata** could include: timestamp, event, user_id, page, session_id, etc.
- Consider **time bucketing** (e.g., day/week) added as a metadata field for temporal filtering.
- For groupings, you might aggregate events by user_id or session_id and create one Document per group with summarized page_content.

Document Structure Recap

Regardless of the method, the goal is to convert raw JSON into a consistent **Document** format that your RAG pipeline expects:

- **page_content**: the text that will be embedded (either pretty-printed JSON or a human-readable summary)
- **metadata**: key fields for filtering, citations, and tracing (ids, types, timestamps, source paths)

Once you have Documents, you can **split** (if long), **embed**, and **store** them in your **vector database**.

When to Use Which Approach?

- **JSONLoader + jq_schema**: Fast, declarative extraction when your target subtree is clear (e.g., employees, orders, items). Great for 1-to-1 record → Document.
- **Custom processing**: When you need transformations, redaction, normalization, rollups/summaries, cross-record context, or custom metadata fields.

Often, a hybrid works best: use jq_schema to isolate the right records, then post-process each record into your preferred text/metadata format.

Video 9: SQL Databases Parsing and Processing
Video Order: 9/9
Topics: SQLite setup, Creating tables, Inserting records, LangChain SQLDatabase & SQLDatabaseLoader, Converting SQL to Documents, Metadata enrichment, Table joins
Difficulty: Beginner → Intermediate

Content
Hello guys.

So we are going to continue the discussion with respect to **pre-processing and parsing**. Already in our previous video, we completed **JSON** pre-processing strategies. We saw multiple examples—how to create JSON files and data, how to read JSON, and finally convert it into a **Document** along with **metadata**. We also saw how to do **custom JSON** pre-processing to create the Document structure itself. Right. So all these specific things have been discussed.

Now one more problem statement that we are going to work on. See, in **data ingestion and parsing**, what are we trying to do? If we have a PDF file, a Word doc, a TXT file, a JSON file—we learned techniques to read those files, extract text/data, and convert it into a **Document** structure. In the Document structure, you have two important things: **content** and **metadata**. And after this, further you can **divide into chunks**. Sometimes the inbuilt functions you use will also internally divide the data into chunks.

In this particular video, what we will be doing is specifically working with a **SQL database** as a data source. Because whenever we talk about SQL databases, we are basically talking about **structured data**. (If I talk about a NoSQL database like MongoDB, in short we are talking about JSON—key-value pairs.) From a SQL database, how can we read the data and **convert it into a Document structure**? That is what we will see.

Creating a Sample SQLite Database

For learning, I'll make it simple using **SQLite**.

1. **Imports & setup**

- import sqlite3
- import os
- Create directory: data/databases/ using os.makedirs(..., exist_ok=True).

2. **Create database file**

- Path: data/databases/company.db
- Connection: sqlite3.connect(path)
- Cursor: con.cursor()

3. **Create tables**

- **employees**: (id INTEGER PRIMARY KEY, name TEXT, role TEXT, department TEXT, salary REAL)
- **projects**: (id INTEGER PRIMARY KEY, name TEXT, status TEXT, budget REAL, lead_id INTEGER)

4. **Insert sample data**

- Employees (list of tuples):
    - (1, "John Doe", "Senior Developer", "Engineering", 120000.0)
    - (2, "Jane Smith", "Data Scientist", "AI", 135000.0)
    - (3, "Mike Johnson", "Project Manager", "PMO", 110000.0)
    - (4, "Sarah Williams", "QA Engineer", "Quality", 90000.0)
- Projects (list of tuples):
    - (101, "RAG System", "In Progress", 250000.0, 1)
    - (102, "Data Platform", "Completed", 400000.0, 2)
    - (103, "Mobile App", "In Progress", 180000.0, 3)
    - (104, "Testing Revamp", "Planned", 120000.0, 2)

Use cursor.executemany(...) to insert, con.commit() to save, and close the connection.

Database Content Exploration (LangChain Utilities)

We'll use LangChain community utilities:

- from langchain_community.utilities import SQLDatabase
- from langchain_community.document_loaders import SQLDatabaseLoader

Method 1: SQLDatabase utility

- Initialize: db = SQLDatabase.from_uri("sqlite:///data/databases/company.db")

- Explore:
    - db.get_usable_table_names() → e.g., {employees, projects}
    - db.get_table_info() → schema DDL + sample rows

This quickly confirms the tables, schema, and a few rows.

Custom SQL → Document Conversion (with Metadata)

Our main aim is reading from the database and turning it into **Documents**.

We'll write a helper:

sql_to_documents(db_path: str) -> List[Document]

Steps:

1. con = sqlite3.connect(db_path); cur = con.cursor()
2. **List tables**: cur.execute("SELECT name FROM sqlite_master WHERE type='table'"); tables = cur.fetchall()
3. For each table:
    - **Column info**: PRAGMA table_info(table_name) → column names
    - **Rows**: SELECT * FROM table_name → fetch all
    - Build a **page_content** string summarizing table name, columns, total records, and include a few **sample rows** for context
    - Create a **Document**:
        - page_content: assembled summary + samples
        - metadata: { "source": db_path, "table": table_name, "record_count": len(rows), "type": "sql_table" }
    - Append to documents
4. (Optional) **Relationships**: run a join to capture cross-table context, e.g.:
5. SELECT e.name AS employee_name, e.role, p.name AS project_name, p.status
6. FROM employees e
7. JOIN projects p ON e.id = p.lead_id;

Convert the result into a human-readable **relationship** Document with metadata like { "source": db_path, "type": "employee_project_join" }.

8. Return the documents list.

When you call it with "data/databases/company.db", you'll see a couple of Documents—one per table—and an extra Document for the **join**. Each Document's page_content includes table summaries and sample rows. From here, you can apply a **text splitter** (e.g., **RecursiveCharacterTextSplitter**) to produce chunked Documents ready for embeddings.

Why Metadata Matters (again)

Adding fields like table, record_count, type, and source enables:

- Filtering answers by table or record type
- Better traceability/citations back to the DB
- Targeted retrieval (e.g., prefer join-derived context for cross-entity questions)

Wrap-up

In this video, we:

- Created a **SQLite** database with employees and projects
- Inserted sample data
- Used LangChain's SQL utilities to inspect schema & rows
- Built a **custom SQL → Document** pipeline, including a **join-based** relationship Document
- Prepared the output for **chunking → embeddings → vector store**

This is a foundational pattern you can adapt to Postgres/MySQL/etc. by switching the DB URI and SQL queries. In the next video, we'll continue expanding ingestion strategies and wire everything into the embedding/vector database steps.

Thank you. Take care.