**Text Splitters in lang chain**

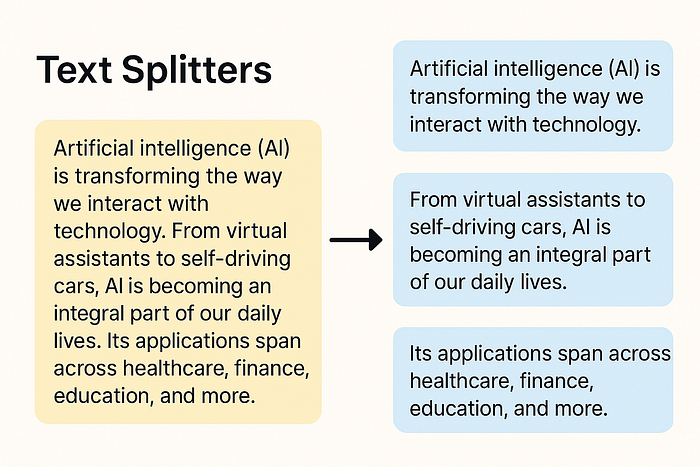
<https://chunkviz.up.railway.app/> I splitting demos

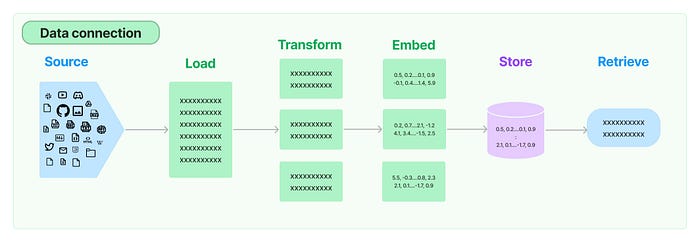
**What is Text Splitting?**

Text splitting is the process of breaking a long document into smaller, easier-to-handle parts. Instead of giving the entire document to an AI system all at once — which might be too much to process — text splitting helps divide the content into chunks of a manageable size.

These chunks are usually based on sentences, paragraphs, or character limits and sometimes include some overlap so the system doesn’t lose the meaning that flows from one part to the next.

Press enter or click to view image in full size





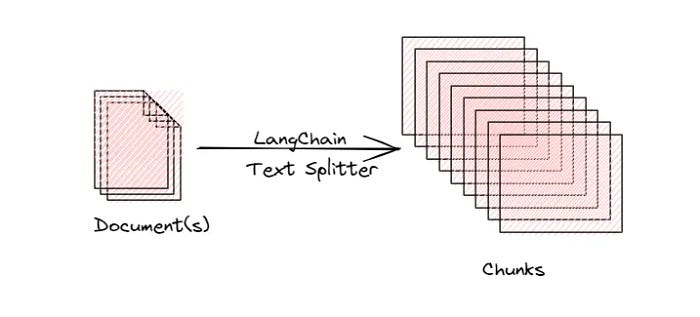
**Need of Splitting Text**

1. LLMs have a limited number of input tokens. Feeding an entire long document may exceed this limit.
2. RAG systems rely on vector similarity search to retrieve relevant text chunks from a document database. Without splitting, the entire document is embedded as one large chunk, which reduces retrieval accuracy.
3. Chunking also affects the embedding quality. Smaller chunks reduce semantic noise and improve the embedding quality.

Now that we understand the need for text splitting, let us explore the different ways we can split the data.

Welcome to Part 10 of the LangChain series! So far, we’ve explored various foundational components — from Models and Prompts to Document Loaders. In this blog, we’ll take a deep dive into a critical yet often overlooked piece of the puzzle — **Text Splitters**.

Press enter or click to view image in full size



**🔍 Why Do We Need Text Splitters?**

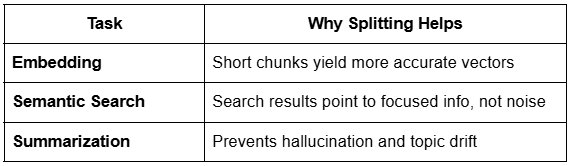
When working with long documents, feeding the entire content to an LLM isn’t feasible due to context length limitations. **Text splitters** help by dividing large text data into smaller, manageable chunks.

But splitting isn’t just about dividing blindly. It’s about **preserving context while minimizing loss of meaning** — which is essential in applications like RAG (Retrieval-Augmented Generation).

**✨ Advantages of Using Text Splitters**

**Overcoming model limitations:** Many embedding models and language models have maximum input size constraints. Splitting allows us to process documents that would otherwise exceed these limits.

**Downstream tasks:**Text Splitting improves nearly every LLM-powered task:



**Optimizing computational resources:** Working with smaller chunks of text can be more memory-efficient and allow for better parallelization of processing tasks.

**📚 Types of Text Splitters in LangChain**

LangChain provides various strategies for splitting documents into chunks, depending on the use case and content structure. Below are the main types of text splitters:

**Length-Based Splitters:**

Splits text into fixed-size chunks based on the number of characters or tokens, regardless of sentence or semantic boundaries.

**Example:**  
Split a 1200-character document into chunks of 400 characters each.

from langchain.text\_splitter import CharacterTextSplitter  
  
splitter = CharacterTextSplitter(chunk\_size=400, chunk\_overlap=50)  
chunks = splitter.split\_text(document\_text)

**🧠 Understanding chunk\_overlap:**  
chunk\_overlap defines how much of the previous chunk is included in the next one.

* ✅ **Ideal chunk overlap for RAG**: 10–20% of chunk size
* 🔻 **Too low overlap**: May **lose context** across chunks
* 🔺 **Too high overlap**: Preserves context but **increases the number of chunks** → higher computation and latency

**Advantages:**  
✅ Simple and fast to implement  
✅ Works well for uniform, unstructured text  
✅ Efficient with large datasets

**Disadvantages:**  
❌ May break sentences in the middle  
❌ Can lead to loss of semantic meaning  
❌ Not ideal for content with rich structure

In LangChain we have different TextSplitter classes to split the text. Firstly, install the required package using the command: pip install langchain\_text\_splitter . For the examples below, we have used text data from a sample PDF. So we use a PyPDFLoader document loader to load the data. And then perform splitting.

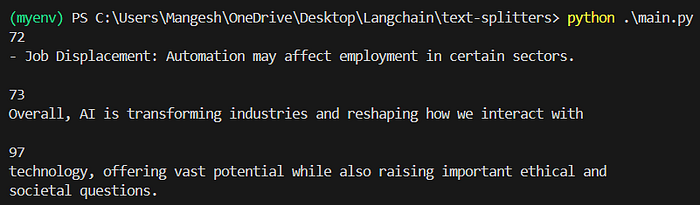
**Text Splitting based on Length**

Here the splitting is done based on a predefined number of chunk\_size . Below is an example.

from langchain\_community.document\_loaders import PyPDFLoader   
from langchain\_text\_splitters import CharacterTextSplitter  
  
loader = PyPDFLoader("AI\_Introduction.pdf")  
documents = loader.load()  
  
text = documents[0].page\_content  
# print(len(text))  
  
text\_splitter = CharacterTextSplitter(  
 separator="\n",  
 chunk\_size=100,   
 chunk\_overlap=0  
)  
  
# chunks = text\_splitter.split\_documents(documents)  
# print(type(chunks[0]))  
  
chunks = text\_splitter.split\_text(text)  
# print(len(chunks))  
  
for chunk in chunks[22:]:  
 print(len(chunk))  
 print(chunk)  
 print()

We can directly split the Document objects using the method split\_documents . Also, we can perform a split on text using the split\_text method. Output is a list containing the resultant chunks.

Press enter or click to view image in full size



Output of Text Splitting based on Length

The chunk does not consist exactly of 100 characters because of what separator we have used. ext will be split only at new lines since we are using the new line (“\n”) as the separator. If any chunk has a size more than 100 but no new lines in it, it will be returned as such.

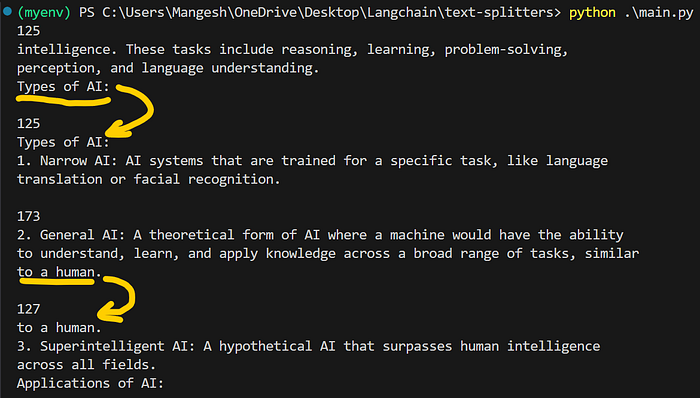
**What is chunk\_overlap ?**

chunk\_overlap is one of the properties of the TextSplitter class (see in the code example above) that specifies the number of characters that are repeated between consecutive chunks when splitting text.

This helps in preserving the semantic meaning between two adjacent chunks by not creating a strict partition between them. Let us code one example for this.

from langchain\_community.document\_loaders import PyPDFLoader   
from langchain\_text\_splitters import CharacterTextSplitter  
  
loader = PyPDFLoader("AI\_Introduction.pdf")  
documents = loader.load()  
  
text = documents[0].page\_content  
  
text\_splitter = CharacterTextSplitter(  
 separator="\n",  
 chunk\_size=200,   
 chunk\_overlap=20  
)  
  
chunks = text\_splitter.split\_text(text)  
  
for chunk in chunks[1:5]:  
 print(len(chunk))  
 print(chunk)  
 print()

Press enter or click to view image in full size



Output of chunk\_overlap

Since we have set chunk\_size=20 , 20 characters are shared between two consecutive chunks.

**Text-Structure-Based Splitters:**

These splitters leverage the **natural structure** of the text — such as newlines, paragraphs, or sentence breaks — to create semantically meaningful chunks.

This technique assumes that **every text inherently follows a structure**:

Text is inherently hierarchical:

📄 Document → 📝 Paragraphs → ✍️ Sentences → 🔤 Words

By following this structure, we can split documents in a way that preserves natural language flow and meaning.

**Example:**  
Split text first by double newlines (paragraphs), then by single newlines (sentences).

from langchain.text\_splitter import RecursiveCharacterTextSplitter  
  
splitter = RecursiveCharacterTextSplitter(  
 chunk\_size=500,  
 chunk\_overlap=50,  
 separators=["\n\n", "\n", ".", " ", ""]  
)  
chunks = splitter.split\_text(document\_text)

**Advantages:**  
✅ Maintains semantic coherence  
✅ Reduces sentence-breaking issues  
✅ More aligned with a human-readable format

**Disadvantages:**  
❌ Performance depends on the presence of structured formatting (newlines, periods, etc.)  
❌ Less control over exact chunk size compared to fixed-length methods

**Text-structured based**

This is the most widely used kind of text splitter, so we will focus more on understanding this. As a base, this uses the fact that text data is organised in some hierarchical structure. The hierarchy can be represented as:

1. The highest level is *Paragraphs*.
2. Next comes *Sentences* that make a paragraph.
3. *Words* combine to phrase a sentence.
4. Words are made up of *Characters*.

We use RecursiveCharacterTextSplitter class in LangChain to split text recursively into smaller units, while *trying*to keep each chunk size in the given limit.

**How it works?**

* Start from the highest level, e.g., paragraphs (separated by \n\n).
* If the paragraph is too long (exceeds the chunk\_size), try to split it into sentences (e.g., using . or \n).
* If that’s still too large, go down to smaller units like words or characters.
* The splitting recurses downward until it produces chunks that are:
* Within the size limit (chunk\_size)
* As semantically meaningful as possible.

Let us code it now!

from langchain\_community.document\_loaders import PyPDFLoader   
from langchain\_text\_splitters import RecursiveCharacterTextSplitter  
  
loader = PyPDFLoader("sample.pdf")  
documents = loader.load()  
  
text = documents[0].page\_content  
  
text\_splitter = RecursiveCharacterTextSplitter(  
 separators=["\n\n", "\n", ".", " ", ""], # Paragraph → Line → Sentence → Word → Character  
 chunk\_size=1000,  
 chunk\_overlap=0  
)  
  
chunks = text\_splitter.split\_text(text)  
  
for chunk in chunks:  
 print(chunk)  
 print()

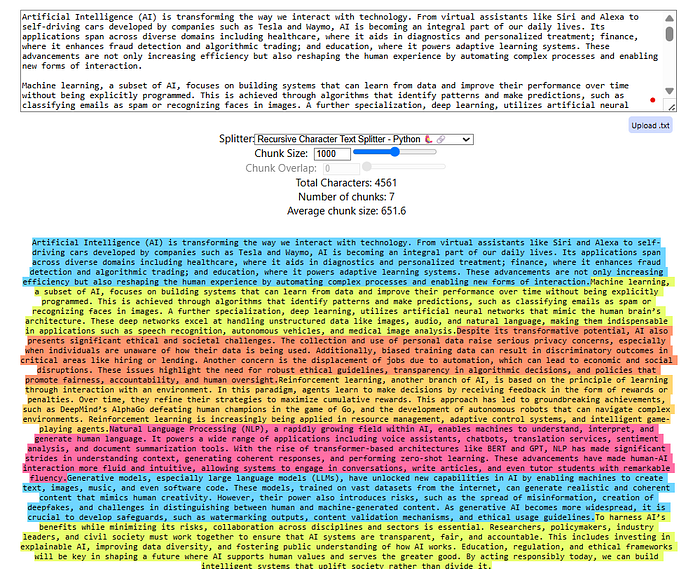
This is an amazing tool to visualize the text splits for better understanding:

**[ChunkViz](https://chunkviz.up.railway.app/?source=post_page-----3a958eea2797---------------------------------------" \t "_blank)**

[Web site created using create-react-app](https://chunkviz.up.railway.app/?source=post_page-----3a958eea2797---------------------------------------" \t "_blank)

[chunkviz.up.railway.app](https://chunkviz.up.railway.app/?source=post_page-----3a958eea2797---------------------------------------" \t "_blank)

Press enter or click to view image in full size



Output of RecursiveCharacterTextSplitter

In separators, we can define complex patterns to split text more efficiently. Play with the text split visualizer by experimenting with different parameter values of chunk\_size , chunk\_overlap , etc.

**Document Structure-Based Splitters**

These splitters are designed to handle documents that follow a **non-standard structure**, such as **code files, markdown files, legal documents**, etc., where traditional paragraph or sentence splitting doesn’t apply.

**Use Case:**

Whenever we want to split a text that is **not regular prose** (like English, Hindi, etc.), we use this splitter.  
For example:

* Python code files
* Markdown documents
* HTML/XML files
* Log files

These formats often use symbols or syntactic patterns like ###, def, or - as logical breakpoints.

**How It Works:**  
We still use the **RecursiveCharacterTextSplitter**, but the only difference is:

✅ We define **custom separators** based on the file structure.

For example, a markdown splitter may look like this:

from langchain.text\_splitter import RecursiveCharacterTextSplitter,Language  
  
text = """  
# Project Name: Smart Student Tracker  
  
A simple Python-based project to manage and track student data, including their grades, age, and academic status.  
  
  
## Features  
  
- Add new students with relevant info  
- View student details  
- Check if a student is passing  
- Easily extendable class-based design  
  
  
## 🛠 Tech Stack  
  
- Python 3.10+  
- No external dependencies  
  
  
## Getting Started  
  
1. Clone the repo   
 ```bash  
 git clone https://github.com/your-username/student-tracker.git  
  
"""  
  
# Initialize the splitter  
splitter = RecursiveCharacterTextSplitter.from\_language(  
 language=Language.MARKDOWN,  
 chunk\_size=200,  
 chunk\_overlap=0,  
)  
  
# Perform the split  
chunks = splitter.split\_text(text)

These custom separators ensure that the split aligns with **document-specific structure**, not general language formatting.

**Advantages:**  
✅ Tailored to specialized document formats (e.g., code, markdown)  
✅ Preserves structural hierarchy when configured correctly

**Disadvantages:**  
❌ Requires domain knowledge to define meaningful separators  
❌ Might fail if the document uses inconsistent formatting

**Semantic Meaning-Based Splitters**

These splitters use **sentence embeddings and cosine similarity** to break text at **meaningful semantic boundaries** — ensuring that each chunk captures a cohesive idea.

**🧠 How It Works Internally**

1. The input text is first **split into sentences**
2. Each sentence is converted into an **embedding vector**
3. **Cosine similarity** is calculated between each pair of neighboring sentence embeddings
4. A **standard deviation** is computed from all the similarity scores
5. Based on a **threshold**, the splitter identifies points where the similarity drops significantly
6. **The new chunk starts from the point** where the similarity deviates from the standard deviation beyond the set limit

from langchain.text\_splitter import SemanticChunker  
from langchain.embeddings import HuggingFaceEmbeddings  
  
embeddings = HuggingFaceEmbeddings(model\_name="all-MiniLM-L6-v2")  
splitter = SemanticChunker(embeddings, breakpoint\_threshold\_type="standard\_deviation\_z\_score", breakpoint\_threshold\_amount=1.5)  
  
chunks = splitter.split\_text(long\_text)

**📏 Parameter:**breakpoint\_threshold\_amount

This is the sensitivity setting that controls how aggressively the text is split:

breakpoint\_threshold\_amount = 1:

* **High sensitivity:**
* Even small drops in similarity → trigger a split
* 🔻 Result: More chunks, highly focused

breakpoint\_threshold\_amount = 3:

* **Low sensitivity:**High chance majority of sentences combine because we are giving much space from the standard deviation.
* More tolerance before splitting
* 🔺 Result: Fewer chunks, broader content span

The higher the threshold, the more sentences get grouped together.

from langchain.text\_splitter import SemanticChunker  
from langchain.embeddings import HuggingFaceEmbeddings  
  
embeddings = HuggingFaceEmbeddings(model\_name="all-MiniLM-L6-v2")  
splitter = SemanticChunker(embeddings, breakpoint\_threshold\_type="standard\_deviation\_z\_score", breakpoint\_threshold\_amount=1.5)  
  
chunks = splitter.split\_text(long\_text)

**Advantages:**  
✅ Semantically meaningful splits  
✅ Ideal for Q&A, summarization, and semantic search  
✅ Reduces context dilution and chunk overlap needs

**Disadvantages:**  
❌ Requires embedding model (adds computational cost)  
❌ Slower than structural/length-based methods  
❌ Complex to tune threshold without experimentation

**Some others splitters**

Any NLP task involving a long document might need to be preprocessed or transformed to improve the accuracy or efficiency of the task at hand. Text Splitter comes in handy when it comes to breaking down huge documents into chunks that will enable analysis at a more granular level. LangChain provides the user with various options to transform the documents by chunking them into meaningful portions and then combining the smaller chunks into larger chunks of a particular size with overlap to retain the context. So the focus will be on how the text is split and how the chunk size is measured.

1. **Character Text Splitter:**This is the simplest method of splitting the text by characters which is computationally cheap and doesn't require the use of any NLP libraries. Here the text split is done on characters and the chunk size is measured by the number of characters.

from langchain.text\_splitter import CharacterTextSplitter  
  
text\_splitter = CharacterTextSplitter(  
 chunk\_size=1000,  
 separator="\n\n",  
 chunk\_overlap=100,  
 length\_function=len,  
 is\_separator\_regex=False  
)  
  
texts = text\_splitter.create\_documents(texts)

The parameter ***chunk\_overlap***helps in retaining the semantic context between the chunks. The metadata can also be passed along with the documents.

**2.** **NLTK Text Splitter:**When we want to focus more on the nature of the context, we might end up using sentence chunking. The basic approach to sentence chunking is **text.split(“.”).**However, LangChain has a better approach that uses NLTK tokenizers to perform text splitting. Here the text split is done on NLTK tokens and the chunk size is measured by the number of characters.

from langchain.text\_splitters import NLTKTextSplitter  
  
text\_splitter = NLTKTextSplitter(chunk\_size=1000, chunk\_overlap = 100)  
  
texts = text\_splitter.split\_text(text)

**3. Spacy Text Splitter:**Another alternative to NLTK is the spaCy tokenizer which offers a more sophisticated sentence segmentation feature that separates the texts into chunks while preserving the context in a better way. Here the text split is done on spaCy tokens and the chunk size is measured by the number of characters.

from langchain.text\_splitter import SpacyTextSplitter  
  
text\_splitter = SpaCyTextSplitter(chunk\_size=1000, chunk\_overlap = 100)  
  
texts = text\_splitter.split\_text(text)

**4. Recursive Character Text Splitter:**This type of text splitter comes into the picture when the text exceeds the chunk length and there is no separator to chunk the text. This method uses a set of separators like [“\n\n”, “\n”, “ “, “”] and splits the text into smaller chunks in an iterative manner until the desired chunk is achieved. The resulting chunks will not be of the same size yet of similar sizes. Here the text split is done on the list of characters and the chunk size is measured by the number of characters.

from langchain.text\_splitter import RecursiveCharacterTextSplitter  
  
text\_splitter = RecursiveCharacterTextSplitter(  
 separators=["\n\n", "\n", ' ', ''],  
 chunk\_size = 1200,  
 chunk\_overlap = 100,  
 length\_function = len\_fun,  
 is\_separator\_regex =False  
)  
  
chunk\_list = text\_splitter.create\_documents(texts)

Note: The split first happens at “\n\n”, if the chunk size exceeds, it will move to the next separator, if it still exceeds, it will move to the next separator which is “ “ and so on.

**5. Splitting on tokens:**Handling token limits in language models is pivotal for seamless operations and optimal performance. It is a good practice to count the number of tokens after getting the chunks.

Tiktoken was created by OpenAI and is a fast BPE. It can be used to track the number of tokens used and better suits OpenAI models. Here the text split is done on the characters passed in and the chunk size is measured by the tiktoken tokenizer.

pip install tiktoken

from langchain.text\_splitter import CharacterTextSplitter  
  
text\_splitter = CharacterTextSplitter.from\_tiktoken\_encoder(  
 separator="\n\n",   
 chunk\_size=1200,   
 chunk\_overlap=100,   
 is\_separator\_regex=False,  
 model\_name='text-embedding-3-small',  
 encoding\_name='text-embedding-3-small',   
)  
  
doc\_list = text\_splitter.create\_documents([text])  
  
doc\_list

The ***model\_name***refers to the model used for calculating the tokens. The split text can also be converted to a list of documents.

from langchain.docstore.document import Document  
  
doc\_list = []  
  
for line in line\_list:  
 curr\_doc = Document(page\_content=line, metadata={"source": filepath})  
 doc\_list.append(curr\_doc)  
  
doc\_list

**6. Sentence Transformers Token Text Splitter:**This type is a specialized text splitter used with sentence transformer models.

from langchain.text\_splitters import SentenceTransformersTokenTextSplitter   
  
splitter = SentenceTransformersTokenTextSplitter(  
 tokens\_per\_chunk=64,  
 chunk\_overlap=0,  
 model\_name='intfloat/e5-base-v2')  
  
  
text\_token\_count = splitter.split\_text(text=text)  
print(text\_token\_count)

**7. Code Splitter:**This type lets you split the code and it comes with multiple language options like Python, java, Latex, HTML, scala, c, and a lot more.

from langchain\_text\_splitters import Language, RecursiveCharacterTextSplitter  
  
text\_splitter = RecursiveCharacterTextSplitter.from\_language(  
 language = Language.PYTHON,  
 chunk\_size=50,  
 chunk\_overlap =10  
)  
  
text\_splitter.create\_documents(texts = [python\_code])

We can even find the type of separator used for the given language.

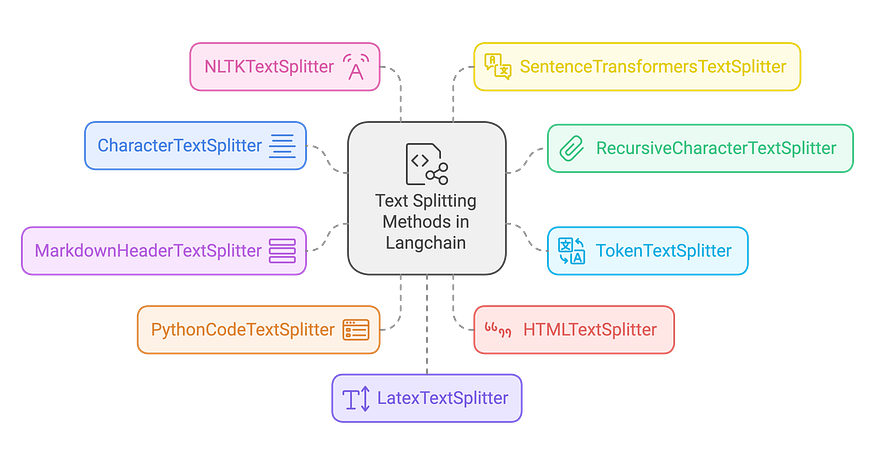
RecursiveCharacterTextSplitter.get\_separators\_for\_language(Language.PYTHON)

And that’s not done there yet. Langchain has a lot of options to perform semantic chunking, token splitters for KoNLPY, Hugging Face, and more.

That’s all about text splitters. Experimenting with a range of chunk sizes will help in bringing a balance between preserving context and maintaining accuracy. Keep in mind that one solution will not work always, so get started with experimentation.

Now that we know how to chunk the text, let's see how we can embed the chunks in our next article. Thanks for reading.

**Some other NLP Based Splitters**



**Introduction:**

In the rapidly evolving world of Natural Language Processing (NLP), improving AI model outputs with relevant and concise information is key to enhancing user experiences. One such approach is Retrieval-Augmented Generation (RAG), where the preprocessing step of **text splitting** plays a pivotal role in achieving high accuracy. Text splitting divides large text into smaller, more manageable chunks, enabling AI systems to process and retrieve relevant content efficiently. Langchain provides an array of text splitting strategies, tailored for diverse document structures and NLP applications. In this article, we’ll discuss some essential text splitters in Langchain, highlight their specific use cases, and demonstrate their functionality with code samples.

**1. Character-Driven Splitting: Simple Yet Effective**

**CharacterTextSplitter** is one of Langchain’s simplest tools for dividing text into chunks based on a fixed character count. It is ideal for straightforward tasks where the document structure is uniform.

**Best for:**

* Homogeneous documents where each chunk is relatively equal in size.
* Simple scenarios where detailed control over chunking isn’t necessary.

from langchain.text\_splitter import CharacterTextSplitter  
  
text = "This is an example text that will be split into manageable chunks. Each chunk will have a maximum length defined."  
  
splitter = CharacterTextSplitter(  
 separator=" ",  
 chunk\_size=500, # 500 characters per chunk  
 chunk\_overlap=100 # 100 characters overlap  
)  
  
chunks = splitter.split\_text(text)  
print(chunks)

Here, the splitter creates chunks of 1000 characters, ensuring a 200-character overlap, using double newlines as a separator to avoid splitting mid-paragraphs.

**2. Flexibility in Chunking with Recursive Splitting**

When more flexibility is required, **RecursiveCharacterTextSplitter** comes in handy. It allows splitting based on multiple separator patterns, progressively refining the chunking logic.

**Best for:**

* Handling documents with varied structures.
* Scenarios requiring optimal chunking based on specific delimiters.

from langchain.text\_splitter import RecursiveCharacterTextSplitter  
  
text = "This is a document with different types of delimiters. We will use recursive splitting to handle it."  
  
splitter = RecursiveCharacterTextSplitter(  
 separators=["\n\n", "\n", " ", ""], # Multiple delimiters to ensure efficient chunking  
 chunk\_size=400, # Chunk size adjusted for flexibility  
 chunk\_overlap=50 # Overlap of 50 characters for maintaining context  
)  
  
chunks = splitter.split\_text(text)  
print(chunks)

This splitter works by first trying to split the text using double newlines, then falling back to single newlines, spaces, and finally individual characters if necessary.

**3. Token-Based Chunking for Model-Sensitive Tasks**

For language models that require processing based on token limits, **TokenTextSplitter** is the ideal solution. It ensures that the chunk sizes are optimized for models like GPT-3, which have token limits that can impact processing.

**Best for:**

* Ensuring chunks fit within token constraints for language models.
* Handling documents where token limits are crucial.

from langchain.text\_splitter import TokenTextSplitter  
  
text = "This document is being split based on token limits."  
  
splitter = TokenTextSplitter(  
 encoding\_name="cl100k\_base",  
 chunk\_size=80, # Split text into chunks of 80 tokens  
 chunk\_overlap=20 # 20-token overlap for context  
)  
  
chunks = splitter.split\_text(text)  
print(chunks)

This splitter operates based on tokens rather than characters, crucial for high-performance models that require precise token management.

**4. Splitting Based on Markdown Headers**

**MarkdownHeaderTextSplitter** is perfect for splitting Markdown documents while maintaining their structural integrity. This method respects headers and organizes chunks based on the document hierarchy.

**Best for:**

* Markdown documentation.
* Projects where content must be divided based on sections, subsections, or headers.

from langchain.text\_splitter import MarkdownHeaderTextSplitter  
  
markdown\_text = """  
# Title  
## Section 1  
This is some content.  
## Section 2  
More content here.  
"""  
  
splitter = MarkdownHeaderTextSplitter(  
 headers\_to\_split\_on=[("#", "Header 1"), ("##", "Header 2")]  
)  
  
chunks = splitter.split\_text(markdown\_text)  
print(chunks)

This splitter uses the Markdown header hierarchy to create logically coherent text chunks, which is vital for maintaining a clean structure in large Markdown files.

**5. Optimizing Code-Based Splitting**

The **PythonCodeTextSplitter** is optimized for Python code documents, splitting text while respecting programming structures such as functions and classes. This method ensures the integrity of the code structure while facilitating easier analysis or documentation.

**Best for:**

* Python codebases.
* Preserving the structure of code while splitting for analysis or documentation.

from langchain.text\_splitter import PythonCodeTextSplitter  
  
python\_code = """  
def example\_function():  
 print("This is an example.")  
   
class ExampleClass:  
 def \_\_init\_\_(self):  
 self.val = 10  
"""  
  
splitter = PythonCodeTextSplitter(  
 chunk\_size=100,  
 chunk\_overlap=20  
)  
  
chunks = splitter.split\_text(python\_code)  
print(chunks)

This approach ensures that code splits are done logically, respecting function or class boundaries for clearer, more structured output.

**6. Web Content Parsing with HTMLTextSplitter**

For web scraping or processing HTML content, **HTMLTextSplitter** is designed to handle the structure of HTML documents. It preserves the context of HTML tags, which is essential for proper content extraction or analysis.

**Best for:**

* HTML web pages.
* Projects where the HTML structure must be maintained for proper analysis.

from langchain.text\_splitter import HTMLTextSplitter  
  
html\_text = """  
<html>  
<head><title>Example</title></head>  
<body>  
<h1>Main Title</h1>  
<p>Paragraph 1</p>  
</body>  
</html>  
"""  
  
splitter = HTMLTextSplitter(  
 chunk\_size=100,  
 chunk\_overlap=20  
)  
  
chunks = splitter.split\_text(html\_text)  
print(chunks)

This splitter works to preserve the integrity of HTML tags while breaking down content into manageable pieces.

**7. Advanced Linguistic Splitting with NLTK**

For those looking for text segmentation based on linguistic principles, **NLTKTextSplitter** utilizes the Natural Language Toolkit (NLTK) to split text based on sentence or paragraph boundaries.

**Best for:**

* Well-structured, linguistically rich text.
* Scenarios where sentence or paragraph integrity is important.

from langchain.text\_splitter import NLTKTextSplitter  
  
text = "This is a document. It contains multiple sentences. Each sentence is split separately."  
  
splitter = NLTKTextSplitter(  
 chunk\_size=150,  
 chunk\_overlap=30  
)  
  
chunks = splitter.split\_text(text)  
print(chunks)

This method helps maintain linguistic structures, ensuring each chunk makes sense in the context of natural language.

**8. Semantic Splitting with Sentence Embeddings**

**SentenceTransformersTextSplitter** splits text based on sentence embeddings, ensuring that chunks retain semantic coherence, which is ideal for tasks requiring understanding of meaning across chunks.

**Best for:**

* Tasks requiring semantic chunking.
* Projects that depend on preserving context and meaning in each chunk.

from langchain.text\_splitter import SentenceTransformersTextSplitter  
  
text = "This is an example text. We aim to split it based on semantic meaning."  
  
splitter = SentenceTransformersTextSplitter(  
 chunk\_size=100,  
 chunk\_overlap=20  
)  
  
chunks = splitter.split\_text(text)  
print(chunks)

This method guarantees that each chunk is semantically coherent, which is essential for higher-level NLP tasks.

**9. LaTeX Document Processing**

For academic or scientific documents written in LaTeX, **LatexTextSplitter** is designed to split text while respecting LaTeX-specific commands and environments.

**Best for:**

* LaTeX documents in academia or scientific research.
* Maintaining LaTeX formatting and structure in processing.

from langchain.text\_splitter import LatexTextSplitter  
  
latex\_text = r"""  
\documentclass{article}  
\begin{document}  
\section{Introduction}  
This is an introduction section with some text.  
\section{Methodology}  
This section discusses the methods used for processing the data.  
\section{Conclusion}  
The conclusion summarizes the findings.  
\end{document}  
"""  
  
splitter = LatexTextSplitter(  
 chunk\_size=100, # Adjust chunk size as needed  
 chunk\_overlap=20 # Adjust overlap for context retention  
)  
  
chunks = splitter.split\_text(latex\_text)  
print(chunks)

This approach helps to split LaTeX files while keeping the formatting and structure intact for more accurate document processing.

**Conclusion**

Selecting the right text splitting method for your RAG pipeline is crucial for improving model performance and accuracy. Langchain offers several specialized splitters, each tailored to specific document types and use cases. Whether you’re working with plain text, code, Markdown, HTML, or LaTeX, Langchain provides flexible solutions that preserve the structure and meaning of your data.

By understanding the strengths and limitations of each text splitter, you can optimize your document processing and create more efficient RAG applications. Experiment with different techniques, tweak parameters to suit your project needs, and ensure that your AI-powered applications are as effective and precise as possible.