

Ch 07. Text Splitter

※ 상태	완료
● 최종 편집 일시	@2025년 12월 28일 오전 12:03

Introduction

문서 분할 과정이 왜 필요할까?

문서 분할 과정

01. 문자 텍스트 분할(CharacterTextSplitter)

`is_separator_regex`의 역할

02. 재귀적 문자 텍스트 분할(RecursiveCharacterTextSplitter)

`CharacterTextSplitter` vs. `RecursiveCharacterTextSplitter`

`CharacterTextSplitter`

`RecursiveCharacterTextSplitter`

03. 토큰 텍스트 분할(TokenTextSplitter)

`TokenTextSplitter`

`tiktoken`

`TokenTextSplitter`

`length_function` vs. `.fromtiktoken_encoder` vs. `TokenTextSplitter`

1. `length_function` 으로 Token의 길이를 측정하는 경우

2. `.from_tiktoken_encoder(...)` 을 사용하는 경우

3. `TokenTextSplitter` 을 사용해 토큰 단위로 직접 자르는 경우

`spaCy`

`SentenceTransformers`

`NLTK`

`KoNLPy`

`Hugging Face tokenizer`

04. 시멘틱 청커(SemanticChunker)

05. 코드 분할 (Python, Markdown, JAVA, C++, C#, GO, JS, Latex 등)

06. 마크다운 헤더 텍스트 분할(MarkdownHeaderTextSplitter)

`TextSplitter.from_language` vs. `MarkdownHeaderTextSplitter`

07. HTML 헤더 텍스트 분할(HTMLHeaderTextSplitter)

08. 재귀적 JSON 분할(RecursiveJsonSplitter)

Introduction

문서 분할은 RAG (Retrieval-Augmented Generation) 시스템의 두 번째 단계이다.

- 정확하게는 RAG를 위한 사전 작업 중 2번째 단계

1. 문서 로드 (Document Loading)

- PDF, 웹 문서, 텍스트 파일 등을 시스템에 불러옴

2. 문서 분할 (Text Splitting)

- 로드된 긴 문서를 작은 조각으로 나눔

→ 크고 복잡한 원본 문서를 효율적으로 활용하기 위해 **작은 규모의 조각 (e.g. Chunk)**으로 쪼개는 단계

3. 임베딩 생성 (Embedding)

- 각 chunk를 벡터로 변환

4. 벡터 저장 (Vector Store / Indexing)

- 검색 가능하도록 DB에 저장

문서 분할 과정이 왜 필요할까?

1. 핀포인트 정보 검색 (정확성)

- 질문 (Query)에 연관성이 있는 정보만 가져올 수 있다.
- 각각의 단위는 특정 주제나 내용에 초점을 맞추므로, 관련성이 높은 정보를 제공한다.

2. 리소스 최적화 (효율성)

- 원본 문서 전체를 LLM으로 입력하게 되면 비용도 많이 발생하고 관계 없는 정보도 포함한 답변을 뱉게 된다.
- 할루시네이션으로 이어져 생뚱맞은 답변을 할 수도 있다

؟ 그러면 문서 선별 과정에서 우리가 얻으려는 정보를 사전에 명확하게 정의할 수 있어야 할까?

쿼리가 명확하면 좋지만 실제 쿼리는 “구글이 앤스로픽에 투자한 금액은 얼마야?”와 같이 불명확한 경우가 대부분이다.

→ 질문에 연관성이 있는 정보만 가져올 수 있다는 것은 이미 문서가 각각의 주제 별로 정보가 모아져 있기 때문에 쿼리를 상정해 정보를 모은다가 아닌 그냥 주제 별로, 연관 있는 것끼리 문서를 저장한다라는 것.

→ 그러면 쿼리가 불명확해도 최대한 연관성 있는 정보만 찾아 리턴 함으로써 불명확한 쿼리에 대한 **검색 품질의 하한선**을 끌어 올려 준다.

(만약 문서 분할을 안 한 채 원본에 대해 바로 쿼리 검색을 진행하게 되면 연관성 없는 정보도 리턴되어 검색 품질을 낮출 것이기 때문)

문서 분할 과정

1. 문서 구조 파악

: 다양한 형식 (PDF, html, e-book....)의 문서에서 구조를 파악한다.

| 구조라 함은 문서의 **header, footer, page num, section title**을 지칭한다

2. 단위 선정

: 문서를 어떤 단위로 나눌지 결정한다.

| 페이지별, 섹션별, 또는 문단별 등 문서의 내용과 목적에 따라 다르다.

3. 단위 크기 설정

: 문서를 몇 개의 토큰 단위로 나눌 것인지를 결정한다.

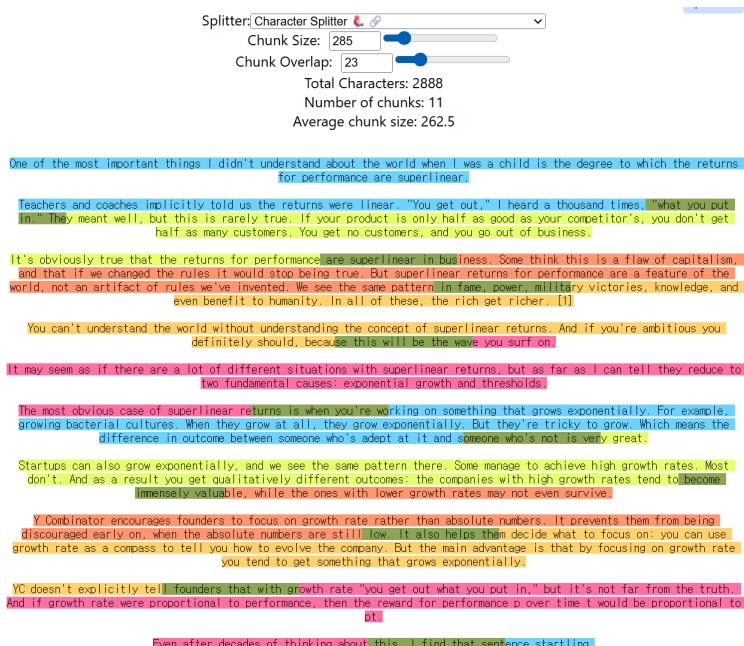
4. 청크 오버랩

: 분할된 끝 부분에서 맥락이 이어질 수 있도록 일부를 겹쳐서(overlap) 분할하는 것.

```
from langchain_text_splitters import RecursiveCharacterTextSplitter

# 단계 2: 문서 분할(Split Documents)
text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=50)
splits = text_splitter.split_documents(docs)
```

<https://chunkviz.up.railway.app/> : Chunk Size와 Chunk Overlap을 시각적으로 볼 수 있다.



01. 문자 텍스트 분할(CharacterTextSplitter)

| 사용 라이브러리 <https://docs.langchain.com/oss/python/integrations/splitters>

? file reading 시 발생하는 encoding 에러는 utf-8 로 읽게끔 바꾸어 해결하였다.

```
# data/appendix-keywords.txt 파일을 열어서 f라는 파일 객체를 생성합니다.
with open("./chapter07Data/appendix-keywords.txt", encoding="utf-8") as f:
    file = f.read() # 파일의 내용을 읽어서 file 변수에 저장합니다.
    ✓ [2] < 10 ms
```

`langchain-text-splitters` 에 내장된 `CharacterTextSplitter` 를 활용해 문자 텍스트 분할을 진행했다.

```
from langchain_text_splitters import CharacterTextSplitter

text_splitter = CharacterTextSplitter(
    # 텍스트를 분할할 때 사용할 구분자를 지정합니다. 기본값은 "\n\n"입니다.
    # separator=" ",
    # 분할된 텍스트 청크의 최대 크기를 지정합니다.
    chunk_size=250,
    # 분할된 텍스트 청크 간의 중복되는 문자 수를 지정합니다.
    chunk_overlap=50,
    # 텍스트의 길이를 계산하는 함수를 지정합니다.
    length_function=len,
    # 구분자가 정규식인지 여부를 지정합니다.
    is_separator_regex=False,
)
```

- `separator` : **분할할 기준 (구분자)**을 설정 (기본값 : `"\n\n"`)

- `chunk_size` 각 청크의 최대 크기를 제한.

 | `len()` 함수를 통해 문자 개수를 기준으로 한 것이다. (`len("abc") == 3`)

 | `chunk_size` 와 `chunk_overlap` 은 `length_function`에 종속적이다

 | → `length`에 대한 정의를 기반으로 조정하는 parameter이기 때문

- `length_function` 텍스트의 길이를 계산하는 함수를 지정

 | 아래와 같이 기준을 무엇으로 둘 것이냐에 따라 다양한 `length_function`은 사용할 수 있다.

- 문자 기준: `len`

- 바이트 기준: `lambda s: len(s.encode("utf-8"))`

- 토큰 기준(토크나이저 필요): `lambda s: len(tokenizer.encode(s))`

- `chunk_overlap` 인접한 청크 간에 n 자의 중복을 허용

- `is_separator_regex`

- True인 경우 : `separator`를 정규식으로 처리 (`Regex Pattern`)

- False인 경우 : `separator`를 일반 문자열로 처리 (`Literal String`)

is_separator_regex의 역할

? `is_separator_regex` 의 역할이 정확하게 무엇일까?

1. True와 False에 따라 어떤 부분에서 차이점을 보이는가?

- `is_separator_regex=False` (기본값)
 - `separator` 를 그대로의 문자열(literal string)로 사용
 - 이 문자열이 정확히 등장하는 위치에서만 분할
- `is_separator_regex=True`
 - `separator` 를 정규식 패턴으로 해석
 - 패턴에 매칭되는 모든 위치에서 분할

2. 정규식으로 분석하는 경우 정확하게 어떤 일이 발생할까?

| 직관적으로 보면 더 유연한 분할이 가능해진다.

예시 (여러 가지 경우에 대한 고려)

- 일반 문자열 방식: `separator="\n\n"`
→ 정확히 줄바꿈 2개일 때만 분할
- 정규식 방식: `separator=r"\n{2,}"`
→ 줄바꿈이 2개 이상 연속되면 모두 분할

3. 그럼 정규식을 활용 하는게 항상 이득인가? → 아니다.

- a. 성능(비용) 문제 : “매칭 비용이 크다”
- b. 과잉 매칭 (over-splitting) 위험 : “너무 잘 쪼개서 문맥을 모르게 된다”
- c. 디버깅 난이도 증가 : “직관적이지 않다”

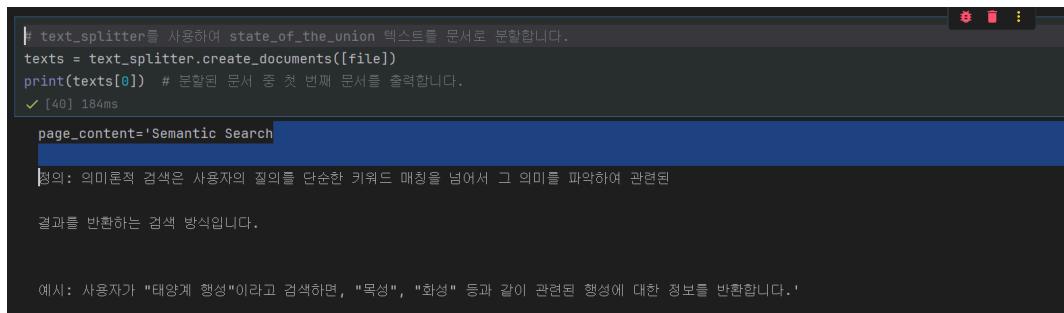
• `create_documents` 메소드를 통해 메타데이터 리스트를 인자로 받아 메타데이터를 주입할 수 있다.

```
metadata = [  
    {"document": 1},  
    {"document": 2},  
] # 문서에 대한 메타데이터 리스트를 정의합니다.  
documents = text_splitter.create_documents(  
    [  
        file,  
        file,  
    ], # 분할할 텍스트 데이터를 리스트로 전달합니다.  
    metadata=metadata, # 각 문서에 해당하는 메타데이터를 전달합니다.  
)  
print(documents[0]) # 분할된 문서 중 첫 번째 문서를 출력합니다.
```

- `split_text()` 메서드를 사용하여 텍스트를 분할한다.

```
# text_splitter를 사용하여 file 텍스트를 분할하고, 분할된 텍스트의 첫 번째 요소를 반환합니다.  
text_splitter.split_text(file)[0]
```

?
구분자로 잘려야 하는데 왜 안 잘리는 것처럼 보이는 부분이 있을까



```
| text_splitter를 사용하여 state_of_the_union 텍스트를 문서로 분할합니다.  
texts = text_splitter.create_documents([file])  
print(texts[0]) # 분할된 문서 중 첫 번째 문서를 출력합니다.  
✓ [40] 184ms  
page_content='Semantic Search'  
  
주의: 의미론적 검색은 사용자의 질의를 단순한 키워드 매칭을 넘어서 그 의미를 파악하여 관련된  
결과를 반환하는 검색 방식입니다.  
  
예시: 사용자가 "태양계 행성"이라고 검색하면, "목성", "화성" 등과 같이 관련된 행성에 대한 정보를 반환합니다.
```

이 부분에서 설정대로라면 Semantic Search 뒤에 `\n\n`이 존재하므로 split 이 적용되어야 하지만 실제로 출력해보면 다음 문단과 합쳐져 나오는 것을 볼 수 있다.

- 이를 통해 만약 split 규칙을 적용하여 잘라낸 뒤에 `chunk_size` 가 작고, 뒤의 chunk를 merge해도 사전에 설정된 최대 `chunk_size` 를 넘지 않는다면 자체적으로 merge process를 거치는 것을 볼 수 있다.
- 정규식을 활용해 중간에 여백이 존재해 separator를 만족하지 않는 경우를 확인해보아도 separator `\n\n` 은 만족하는 것을 볼 수 있다.

```
# 1) 실제로 \n\n가 몇 번 있는지  
print("count('\\\\n\\\\n') =", file.count("\n\n"))  
  
# 2) 원도우 개행인지  
print("count('\\\\r\\\\n') =", file.count("\r\n"))  
print("count('\\\\r\\\\n\\\\r\\\\n') =", file.count("\r\n\r\n"))  
  
# 3) '빈 줄'처럼 보이지만 공백이 있는 패턴  
import re  
print("blank lines with spaces =", len(re.findall(r"\n[ \t]+\n", file)))  
  
✓ [39] < 10 ms  
count('\n\n') = 58  
count('\r\n') = 0  
count('\r\n\r\n') = 0  
blank lines with spaces = 0
```

02. 재귀적 문자 텍스트 분할 (RecursiveCharacterTextSplitter)

사용 라이브러리 <https://docs.langchain.com/oss/python/integrations/splitters>

RecursiveCharacterTextSplitter 를 사용하여 텍스트를 작은 청크로 분할을 진행하였다.

```

text_splitter = RecursiveCharacterTextSplitter(
    # 청크 크기를 매우 작게 설정합니다. 예시를 위한 설정입니다.
    chunk_size=250,
    # 청크 간의 중복되는 문자 수를 설정합니다.
    chunk_overlap=50,
    # 문자열 길이를 계산하는 함수를 지정합니다.
    length_function=len,
    # 구분자로 정규식을 사용할지 여부를 설정합니다.
    is_separator_regex=False,
)

```

| 기본적으로 사용되는 parameter는 이전의 `CharacterTextSplitter` 와 동일한 역할을 수행한다.

- `chunk_size` 각 청크의 최대 크기를 제한.
- `chunk_overlap` 매개변수를 50으로 설정하여 인접한 청크 간에 50자의 중복을 허용합니다.
- `length_function` 텍스트의 길이를 계산하는 함수를 지정
- `is_separator_regex` 매개변수를 False로 설정하여 separator를 정규식이 아닌 일반 문자열로 처리합니다.

CharacterTextSplitter vs. RecursiveCharacterTextSplitter

CharacterTextSplitter

1. 하나의 **separator**로 먼저 나눈다.
2. 잘린 조각들을 `chunk_size` 를 넘지 않게 묶어서 chunk를 만든다.
3. 한 청크가 이미 `chunk_size` 보다 큰데 그 조각 안에는 separator가 없으면 (더 이상 나눌 수 없으면)
→ 더 이상 잘게 못 쪼개고 `chunk_size` 를 초과한 채로 남게 된다.

RecursiveCharacterTextSplitter

- “재귀적”이라는 말 그대로, 여러 단계의 **separator 목록**을 가지고
 1. 문단(`\n\n`)으로 나눠 보고
 2. 그래도 크면 줄바꿈(`\n`)으로 더 나누고
 3. 그래도 크면 공백()으로
 4. 마지막엔 문자 단위까지

이런 식으로 `chunk_size` 를 만족할 때까지 더 작은 단위로 내려가며 쪼갭니다.

그래서 실제로 돌려보면 `CharacterTextSplitter` 에서는 `chunk_size` 보다 큰 청크가 존재하지만 (사실 1번을 제외하고는 모두 초과한다), `RecursiveCharacterTextSplitter` 에서는 청크 크기가 모두 `chunk_size` 보다 작다

| 만약 `chunk_size`를 엄격하게 지키고 싶다면 Recursive를 활용하는 것이 더 적합해보인다.

```

==== CharacterTextSplitter (separator='\n\n') ====
num_chunks = 6
[01] len= 59 | 'F1 Weekend Notebook: Strategy, Tyres, and the Small Details'
[02] len= 294 | 'In modern Formula 1, the headline is often a single lap: pole position, a daring'...
[03] len= 271 | 'Section 1 – Why Tyre Management Looks Boring on TV\nThe tyre is not just a grip g'...
[04] len=1931 | 'Section 2 – Race Radio (One Long Block On Purpose)\nENGINEER: We are boxing this '...
[05] len= 250 | 'Section 3 – The Pit Stop as a System\nA pit stop is a choreography of constraints'...
[06] len= 259 | 'Section 4 – The Lap Time is a Stack of Micro-Choices\nLift two meters earlier, ch'...

==== RecursiveCharacterTextSplitter (default separators) ====
num_chunks = 35
[01] len= 59 | 'F1 Weekend Notebook: Strategy, Tyres, and the Small Details'
[02] len= 109 | 'In modern Formula 1, the headline is often a single lap: pole position, a daring'...
[03] len= 104 | 'But most races are decided in the quiet moments—when the driver saves the front '...

[04] len= 79 | 'chooses the safer undercut, and when a team avoids one extra second in the box.'
[05] len= 50 | 'Section 1 – Why Tyre Management Looks Boring on TV'
[06] len= 111 | 'The tyre is not just a grip generator; it is a time budget. If you spend it too '...
[07] len= 108 | 'survival. If you protect it too much, you never unlock pace. The best drivers ma'...
[08] len= 50 | 'Section 2 – Race Radio (One Long Block On Purpose)'
[09] len= 92 | 'ENGINEER: We are boxing this lap. Tyres are falling off the cliff, keep it tidy '...
[10] len= 84 | "DRIVER: Copy. Front left is gone. I can't rotate mid-corner and I'm sliding on e"...

```

▼ 사용한 글 원문

F1 Weekend Notebook: Strategy, Tyres, and the Small Details

In modern Formula 1, the headline is often a single lap: pole position, a daring pass, or a last-lap defence.

But most races are decided in the quiet moments—when the driver saves the front tyres, when the pit wall chooses the safer undercut, and when a team avoids one extra second in the box.

Section 1 — Why Tyre Management Looks Boring on TV

The tyre is not just a grip generator; it is a time budget. If you spend it too early, your final stint becomes survival. If you protect it too much, you never unlock pace. The best drivers make the compromise invisible.

Section 2 — Race Radio (One Long Block On Purpose)

ENGINEER: We are boxing this lap. Tyres are falling off the cliff, keep it tidy in Sector 3.
 DRIVER: Copy. Front left is gone. I can't rotate mid-corner and I'm sliding on exit.
 ENGINEER: Understood. Brake bias plus one click forward. Manage the fronts, short-shift out of Turn 10.
 DRIVER: The car is snappy in slow speed. I'm losing rear stability in Turn 14.
 ENGINEER: We see that. Differential mid minus two. Target delta is plus 0.4 to the leader.
 DRIVER: Tell me about traffic. I don't want to get stuck behind the Haas again.
 ENGINEER: Next car ahead is on old hards, 2.1 seconds. Use overtake on the main straight only.
 ENGINEER: We are boxing this lap. Tyres are falling off the cliff, keep it tidy in Sector 3.
 DRIVER: Copy. Front left is gone. I can't rotate mid-corner and I'm sliding on exit.
 ENGINEER: Understood. Brake bias plus one click forward. Manage the fronts, short-shift

t out of Turn 10.
DRIVER: The car is snappy in slow speed. I'm losing rear stability in Turn 14.
ENGINEER: We see that. Differential mid minus two. Target delta is plus 0.4 to the leader.
DRIVER: Tell me about traffic. I don't want to get stuck behind the Haas again.
ENGINEER: Next car ahead is on old hards, 2.1 seconds. Use overtake on the main straight only.
ENGINEER: We are boxing this lap. Tyres are falling off the cliff, keep it tidy in Sector 3.
DRIVER: Copy. Front left is gone. I can't rotate mid-corner and I'm sliding on exit.
ENGINEER: Understood. Brake bias plus one click forward. Manage the fronts, short-shift out of Turn 10.
DRIVER: The car is snappy in slow speed. I'm losing rear stability in Turn 14.
ENGINEER: We see that. Differential mid minus two. Target delta is plus 0.4 to the leader.
DRIVER: Tell me about traffic. I don't want to get stuck behind the Haas again.
ENGINEER: Next car ahead is on old hards, 2.1 seconds. Use overtake on the main straight only.

Section 3 — The Pit Stop as a System

A pit stop is a choreography of constraints: wheel gun torque limits, jack release timing, safe-release rules, and the human factor. A 2.1s stop is not “fast”; it is an error-free sequence executed under pressure.

Section 4 — The Lap Time is a Stack of Micro-Choices

Lift two meters earlier, change the line to cool the surface, avoid the dirty air, and choose when to deploy battery. None of these are dramatic alone, but together they decide whether you attack or defend.

03. 토큰 텍스트 분할(TokenTextSplitter)

TokenTextSplitter

실제 언어 모델에서 가장 많이 쓰이는 단위는 토큰으로, `TokenTextSplitter` 는 텍스트를 토큰 수를 기반으로 청크를 생성할 때 유용하다.

tiktoken

`tiktoken` 은 OpenAI에서 만든 빠른 `BPE Tokenizer` 입니다.

`CharacterTextSplitter` 를 사용하여 텍스트를 분할합니다.

- `from_tiktoken_encoder` 메서드를 사용하여 Tiktken 인코더 기반의 텍스트 분할기를 초기화합니다.

```
from langchain_text_splitters import CharacterTextSplitter
```

```
text_splitter = CharacterTextSplitter.from_tiktoken_encoder(  
    # 청크 크기를 300으로 설정합니다.
```

```

        chunk_size=300,
        # 청크 간 중복되는 부분이 없도록 설정합니다.
        chunk_overlap=0,
    )
# file 텍스트를 청크 단위로 분할합니다.
texts = text_splitter.split_text(file)

```

| 마찬가지로 RecursionCharacterTextSplitter를 활용하면 실제로 분할되는 토큰의 크기를 보장할 수 있다.

TokenTextSplitter

- `TokenTextSplitter` 클래스를 사용하여 텍스트를 토큰 단위로 분할합니다.

```

from langchain_text_splitters import TokenTextSplitter

text_splitter = TokenTextSplitter(
    chunk_size=200, # 청크 크기를 10으로 설정합니다.
    chunk_overlap=0, # 청크 간 중복을 0으로 설정합니다.
)

# state_of_the_union 텍스트를 청크로 분할합니다.
texts = text_splitter.split_text(file)
print(texts[0]) # 분할된 텍스트의 첫 번째 청크를 출력합니다.

```

`length_function` vs. `.fromtiktoken_encoder` vs. `TokenTextSplitter`

| 지금까지 토큰을 기준으로 분할할 수 있는 방법을 3가지로 배웠는데 어떤 차이점이 존재하는가

1. `length_function` 으로 Token의 길이를 측정하는 경우

- **분할(자르는 위치):** `\n\n → \n → " " → ""` 같은 “**문자/구분자**” 규칙 (**separator**)을 최대한 지키면서 자른다.
- **길이 계산 (몇 단위인지):** `length_function(text)` 가 반환하는 값 (예: 토큰 수)으로 `chunk_size`, `chunk_overlap` 을 적용

즉, 문단/문장/단어 경계를 최대한 보존하면서 길이의 기준만 `token` 으로 바꾼 사례

장점

- 문맥 보존에 유리
- 모델 입력 길이 기준으로 **대략적인 제어 가능**

한계점

- 직접 `Tokenizer` 를 준비해야 함 (수작업으로 세팅을 해야 한다.)
- “자르는 기준 자체가 토큰”은 아니므로, 토큰 단위로 완전히 깔끔하게 맞추는 건 한계가 있음.

2. `.from_tiktoken_encoder(...)` 을 사용하는 경우

| 1번의 `length_function` 과 유사한 동작을 수행한다.

- **분할 위치:** 문자/구분자 기반(Recursive/Character 방식)
- **길이 계산:** tiktoken을 이용해 “토큰 길이”로 측정

| 다만 `length_function` 을 직접 넣는 대신, `tiktoken` 을 쓰도록 안전하게 세팅해주는 편의 생성자이다.

장점

- 토큰 카운트 구현을 직접 하지 않아도 됨 (1번 방식과의 차이점)
- OpenAI 계열 모델 기준 토큰 계산을 더 안전하게 적용 가능

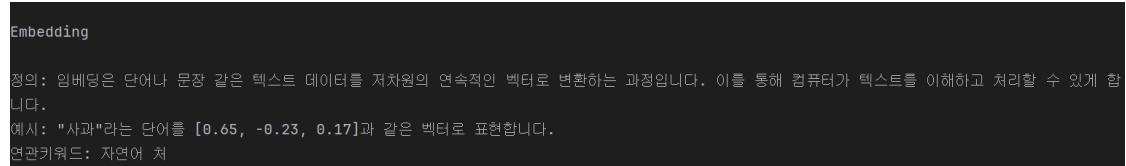
한계점

- 여전히 “분할 위치”는 토큰이 아니라 **구분자/문자 기반**
- 즉, 토큰 윈도우로 정확히 자르는 방식은 아님

3. `TokentTextSplitter` 을 사용해 토큰 단위로 직접 자르는 경우

- **분할 위치 자체가 토큰 단위**
 - 텍스트를 토큰으로 **인코딩** → 토큰 배열을 `chunk_size` 단위로 슬라이스 → **디코딩**
- separator 유지가 우선 되지 않는다.

| 정확히 **N 토큰씩** 청크를 만들기 쉬우나, 문장/단어 중간에서 끊길 수 있다.



장점

- 토큰 수를 매우 정확하게 맞출 수 있음
- 모델 컨텍스트 윈도우 관리(예: 512토큰씩) 같은 작업에 직관적

한계점

- 가독성과 의미 단위 손실 되어 문맥 정보를 잃을 수 있다.
- CJK와 같은 비영어권 텍스트에 대해서 토큰-디코딩 과정이 원문을 완벽히 보존하지 못하는 케이스가 생길 수 있다.

| 실제로 우리의 예시에서도 이러한 경우가 발생했다. (`SentenceTransformersTokenTextSplitter` 사례)

디코딩하는 과정에서 원문을 완벽하게 보존하지 못하였고 해당 원문은 [UNK]으로 표시된 것을 볼 수 있다.

```
# 0번째 청크를 출력합니다.
print(text_chunks[1]) # 분할된 텍스트 청크 중 두 번째 청크를 출력합니다.
```

. 이를 통해 컴퓨터가 텍스트를 이해하고 처리할 수 [UNK] 합니다. [UNK] : " 사과 " 라는 단어를 [0. 65, - 0. 23, 0. 17] 과 [UNK] 벡터로 표현합니다. 연관키워드 : 자연어 처리, 벡터화, 딥러닝 token 정의 : 토큰은 텍스트를 더 작은 [UNK] 분할하는 [UNK] 의미합니다. 이는 일반적으로 단어, 문장, [UNK] 구절일 수 [UNK]. [UNK] : 문장 " 나는 학교에 간다 " 를 " 나는 ", " 학교에 ", " 간다 " 로 분할합니다. 연관키워드 : 토큰화, 자연어 처리, 구문 분석 tokenizer 정의 : 토큰

방법	어디서 자르나	"길이를 무엇으로 재나"	특징
<code>length_function=토큰카운트</code>	구분자/문자 기반	커스텀(토큰 카운트)	구현 자유도 높음, 실수 가능성도 있음
<code>.from_tiktoken_encoder</code>	구분자/문자 기반	tiktoken 토큰	표준/안정 세팅, 보통 이쪽이 더 실무적
<code>TokenTextSplitter</code>	토큰 기반	토큰	토큰 길이 가장 정확, 경계 보존은 약함

spaCy

| Python과 Cython 프로그래밍 언어로 작성된 고급 자연어 처리를 위한 오픈 소스 소프트웨어 라이브러리 Cython이 궁금하다면? <https://ko.wikipedia.org/wiki/%EC%82%AC%EC%9D%B4%EC%8D%AC>
<https://spacy.io/>

- 텍스트가 분할되는 방식: **spaCy tokenizer**에 의해 분할
- chunk size가 측정되는 방법: 문자 수로 측정

```
import warnings
from langchain_text_splitters import SpacyTextSplitter

# 경고 메시지를 무시합니다. (당장은 없애도 별 차이가 없었음)
warnings.filterwarnings("ignore")

# SpacyTextSplitter를 생성합니다.
text_splitter = SpacyTextSplitter(
    chunk_size=200, # 청크 크기를 200으로 설정합니다.
    chunk_overlap=50, # 청크 간 중복을 50으로 설정합니다.
)
```

SentenceTransformers

| SentenceTransformersTokenTextSplitter 는 sentence-transformer 모델에 특화된 텍스트 분할기

사용하고자 하는 sentence transformer 모델의 토큰 윈도우에 맞게 텍스트를 청크로 분할한다.

- ❓ SentenceTransformer (a.k.a. SBERT)란
<https://sbert.net/>

```
https://github.com/huggingface/sentence-transformers
```

- State-of-the-Art Text Embeddings 이라고 주장되는 문장이나 짧은 문단을 고정 길이 벡터 (embedding)로 변환하도록 학습된 Transformer 모델
- <http://www.ukp.tu-darmstadt.de/>에서 개발했다.
- **Embeddings, Retrieval, and Reranking**를 다루기 위해 만든 모델 같다
- 문장의 의미를 벡터로 변환하는데 특화되어 있다.

```
from langchain_text_splitters import SentenceTransformersTokenTextSplitter

# 문장 분할기를 생성하고 청크 간 중복을 0으로 설정합니다.
splitter = SentenceTransformersTokenTextSplitter(chunk_size=200, chunk_overlap=0)
```

```
# 0번째 청크를 출력합니다.
print(text_chunks[1]) # 분할된 텍스트 청크 중 두 번째 청크를 출력합니다.
```

. 이를 통해 컴퓨터가 텍스트를 이해하고 처리할 수 [UNK] 합니다. [UNK] : " 사과 " 라는 단어를 [0. 65, - 0. 23, 0. 17] 과 [UNK] 벡터로 표현합니다. 연관키워드 : 자연어 처리, 벡터화, 딥러닝 token 정의 : 토큰은 텍스트를 더 작은 [UNK] 분할하는 [UNK] 의미합니다. 이는 일반적으로 단어, 문장, [UNK] 구절일 수 [UNK]. [UNK] : 문장 " 나는 학교에 간다 " 를 " 나는 ", " 학교에 ", " 간다 " 로 분할합니다. 연관키워드 : 토큰화, 자연어 처리, 구문 분석 tokenizer 정의 : 토큰

- 앞서 본 예시이다. SentenceTransformersTokenTextSplitter를 활용하게 되면 토큰으로 인코딩 및 디코딩 하는 과정에서 한국어 원문을 완벽하게 복원하지 못하는 모습을 볼 수 있다.

NLTK

Natural Language Toolkit (NLTK)은 Python 프로그래밍 언어로 작성된 영어 자연어 처리(NLP)를 위한 라이브러리와 프로그램 모음

<https://www.nltk.org/>

NLTK tokenizers 를 기반으로 텍스트를 분할하는 데 NLTK를 사용할 수 있다.

NLTK Tokenizer에 대해서는 아래 링크를 통해 볼 수 있다.

<https://www.nltk.org/howto/tokenize.html>

1. 텍스트 분할 방법: NLTK tokenizer 에 의해 분할
2. chunk 크기 측정 방법: 문자 수에 의해 측정
3. 텍스트 데이터의 전처리, 토큰화, 형태소 분석, 품사 태깅 등 다양한 NLP 작업을 수행할 수 있다.

```

from langchain_text_splitters import NLTKTextSplitter

text_splitter = NLTKTextSplitter(
    chunk_size=200, # 청크 크기를 200으로 설정합니다.
    chunk_overlap=0, # 청크 간 중복을 0으로 설정합니다.
)

```

KoNLPy

| KoNLPy(Korean NLP in Python)는 한국어 자연어 처리(NLP)를 위한 파이썬 패키지

<https://konlpy.org/ko/latest/index.html>

- 한국어 NLP 작업에 특화된 패키지. 강력한 대신 연산 비용도 높아진다.
- 한국어 형태소 태깅은 다음 링크를 통해 볼 수 있다
https://docs.google.com/spreadsheets/d/1OGAjUvalBuX-oZvZ_-9tEfYD2gQe7hTGsgUpiIBSXl8/edit?gid=0#gid=0
- KoNLPY에는 **Kkma** (Korean Knowledge Morpheme Analyzer)라는 형태소 분석기가 있어 상세한 분석이 가능하다
- Kkma 이외에도 다양한 모델이 존재한다.

성능 분석

성능 검증은 몇 개의 샘플 문장을 비교하는 것으로 대체합니다.

1. “아버지가방에들어가신다”

이 예시를 통해 띄어쓰기 알고리즘의 성능을 확인해볼 수 있습니다. 이상적인 경우, 이 예시에 대해서는 아버지 + 가방에 + 들어가신다 보다는 아버지가 + 방에 + 들어가신다로 해석하는 것이 더 바람직하겠지요.

Hannanum	Kkma	Komoran	Mecab	Twitter
아버지가방에 들어가 / N	아버지 / NNG	아버지가방에 들어가신다 / NNP	아버지 / NNG	아버지 / Noun
이 / J	가방 / NNG		가 / JKS	가방 / Noun
시ㄴ다 / E	에 / JKM		방 / NNG	에 / Josa
	들어가 / VV		에 / JKB	들어가신 / Verb
	시 / EPH		들어가 / VV	다 / Eomi
	ㄴ다 / EFN		신다 / EP+EC	

KoNLPy 공식 사이트의 성능 분석 표

```

from langchain_text_splitters import KonlpyTextSplitter

```

```

# KonlpyTextSplitter를 사용하여 텍스트 분할기 객체를 생성합니다.
text_splitter = KonlpyTextSplitter()

```

Hugging Face tokenizer

https://huggingface.co/docs/transformers/main_classes/tokenizer

huggingface에서는 다양한 Tokenizer를 제공하고, 이 책의 예시에서는 `pretrainedTokenizer` 중 하나인 `GPT2TokenizerFast`를 사용한다.

```
from transformers import GPT2TokenizerFast

# GPT-2 모델의 토크나이저를 불러옵니다.
hf_tokenizer = GPT2TokenizerFast.from_pretrained("gpt2")

text_splitter = CharacterTextSplitter.from_huggingface_tokenizer(
    # 허깅페이스 토크나이저를 사용하여 CharacterTextSplitter 객체를 생성합니다.
    hf_tokenizer,
    chunk_size=300,
    chunk_overlap=50,
)
# state_of_the_union 텍스트를 분할하여 texts 변수에 저장합니다.
texts = text_splitter.split_text(file)
```

- 따로 Splitter를 생성하는게 아니라 기존에 Langchain에서 제공하는 `CharacterTextSplitter`에서 tokenizer 설정만 huggingface 모델 전용으로 초기화를 해준 것을 볼 수 있다.

`CharacterTextSplitter`는 꽤나 유연하게 커스텀이 가능해보인다.

04. 시멘틱 청커(SemanticChunker)

텍스트를 의미론적 유사성에 기반하여 분할한다. 하지만 이 교재에서 제시되는 예시로는 효과를 보기 어렵다

- 교재에서 활용되는 예시는 구조가 반복되고 미세한 개념의 차이만 존재한다.
- 따라서 유사성으로만 분할을 진행하는 경우 아예 `chunk` 가 하나만 생성이 되거나, 엉뚱한 부분에서 최대 `chunk_size` 초과로 인해 자동으로 잘리는 것처럼 보이는 부분이 존재한다.

reference : https://github.com/FullStackRetrieval-com/RetrievalTutorials/blob/main/tutorials/LevelsOfTextSplitting/5_Levels_Of_Text_Splitting.ipynb

`SemanticChunker`는 LangChain의 실험적 기능 중 하나로, 텍스트를 의미론적으로 유사한 청크로 분할한다.

Breakpoints

SemanticChunker는 문장을 분리할 시점을 두 문장 간 임베딩 차이를 살펴봄으로써 이루어진다. 다시 말해 일종의 breakPoint에 대해 자체적으로 검증을 하는 것이다.

- Percentile 기반의 BreakPoint → 60개의 청크 생성

문장 간의 모든 차이를 계산한 다음, 지정한 백분위수를 기준으로 분리

```
text_splitter = SemanticChunker(  
    # OpenAI의 임베딩 모델을 사용하여 시맨틱 청커를 초기화합니다.  
    OpenAIEmbeddings(),  
    # 분할 기준점 유형을 백분위수로 설정합니다.  
    breakpoint_threshold_type="percentile",  
    breakpoint_threshold_amount=70,  
)
```

2. Standard Deviation 기반의 BreakPoint → 14개의 청크 생성

| 지정한 `breakpoint_threshold_amount` 표준편차보다 큰 차이가 있는 경우 분할

```
text_splitter = SemanticChunker(  
    # OpenAI의 임베딩 모델을 사용하여 시맨틱 청커를 초기화합니다.  
    OpenAIEmbeddings(),  
    # 분할 기준으로 표준 편차를 사용합니다.  
    breakpoint_threshold_type="standard_deviation",  
    breakpoint_threshold_amount=1.25,  
)
```

3. Interquartile 기반의 BreakPoint → 13개의 청크 생성

| 사분위수 범위 (interquartile range)를 사용하여 청크를 분할

```
text_splitter = SemanticChunker(  
    # OpenAI의 임베딩 모델을 사용하여 의미론적 청크 분할기를 초기화합니다.  
    OpenAIEmbeddings(),  
    # 분할 기준점 임계값 유형을 사분위수 범위로 설정합니다.  
    breakpoint_threshold_type="interquartile",  
    breakpoint_threshold_amount=0.5,  
)
```

05. 코드 분할 (Python, Markdown, JAVA, C++, C#, GO, JS, Latex 등)

| CodeTextSplitter를 사용하면 다양한 프로그래밍 언어로 작성된 코드를 분할할 수 있고,

| 이 교재에서는 Python, JS, TS, Markdown, LaTex, HTML, Solidity (블록체인 이더리움 관련 언어), C가 제시되어 있다.

| Splitter는 앞서 제시된 것들을 활용할 수 있고, 이 교재에서는 `RecursiveCharacterTextSplitter`를 사용하여 텍스트를 분할한다.

- `langchain_text_splitters` 모듈에서 `Language` 를 활용해 분할 하려는 코드의 언어를 선택할 수 있다.

```
from langchain_text_splitters import Language
# 지원되는 언어의 전체 목록을 가져옵니다.
[e.value for e in Language]
```

- 사용 가능한 언어 목록

```
['cpp', 'go', 'java', 'kotlin', 'js', 'ts', 'php', 'proto', 'python', 'rst', 'ruby',
'rust', 'scala', 'swift', 'markdown', 'latex', 'html', 'sol', 'csharp', 'cobol', 'c',
'lua', 'perl']
```

- 구조가 반복되기 때문에 파이썬에 대한 Splitter 생성만 정리

- `.from_language` 메소드를 활용하여 `Language` 의 attribute 중 하나인 `PYTHON` 을 활용하는 방식으로 적용하였다.

```
PYTHON_CODE = """
def hello_world():
    print("Hello, World!")

hello_world()
"""

python_splitter = RecursiveCharacterTextSplitter.from_language(
    language=Language.PYTHON, chunk_size=50, chunk_overlap=0
)
```

06. 마크다운 헤더 텍스트 분할 (MarkdownHeaderTextSplitter)

- 문서를 지정된 헤더 집합에 따라 분할하여, 각 헤더 그룹 아래의 내용을 별도의 청크로 관리할 수 있게 한다.

- 먼저 헤더 집합을 선언한 뒤 MarkdownHeaderTextSplitter를 생성해줌으로써 텍스트 분할을 진행할 수 있다.

```
headers_to_split_on = [ # 문서를 분할할 헤더 레벨과 해당 레벨의 이름을 정의합니다.
    (
        "#",
        "Header 1",
    ), # 헤더 레벨 1은 '#'로 표시되며, 'Header 1'이라는 이름을 가집니다.
    (
        "##",
        "Header 2",
    ), # 헤더 레벨 2는 '##'로 표시되며, 'Header 2'라는 이름을 가집니다.
    (
        "###",

```

```

        "Header 3",
    ), # 헤더 레벨 3은 '###'로 표시되며, 'Header 3'이라는 이름을 가집니다.
]

# 마크다운 헤더를 기준으로 텍스트를 분할하는 MarkdownHeaderTextSplitter 객체를 생성합니다.
markdown_splitter = MarkdownHeaderTextSplitter(headers_to_split_on=headers_to_split_on)
# markdown_document를 헤더를 기준으로 분할하여 md_header_splits에 저장합니다.
md_header_splits = markdown_splitter.split_text(markdown_document)
# 분할된 결과를 출력합니다.
for header in md_header_splits:
    print(f"{header.page_content}")
    print(f"{header.metadata}", end="\n=====\n")

```

- `MarkdownHeaderTextSplitter` 는 분할되는 헤더를 출력 청크의 내용에서 제거한다. (기본값)
- 이는 `strip_headers = False` 로 설정하여 비활성화할 수 있다
 - 헤더를 지웠을 때 (기본값)

```

Hi this is Jim
Hi this is Joe
{'Header 1': 'Title', 'Header 2': '1. SubTitle'}
=====
Hi this is Lance
{'Header 1': 'Title', 'Header 2': '1. SubTitle', 'Header 3': '1-1. Sub-SubTitle'}
=====
Hi this is Molly
{'Header 1': 'Title', 'Header 2': '2. Baz'}
=====
```

- 헤더를 남겼을 때

```

# Title
## 1. SubTitle
Hi this is Jim
Hi this is Joe
{'Header 1': 'Title', 'Header 2': '1. SubTitle'}
=====
### 1-1. Sub-SubTitle
Hi this is Lance
{'Header 1': 'Title', 'Header 2': '1. SubTitle', 'Header 3': '1-1. Sub-SubTitle'}
=====
## 2. Baz
Hi this is Molly
{'Header 1': 'Title', 'Header 2': '2. Baz'}
=====
```

- 그룹 내에서 재분할을 원할 경우, 앞서 배운 TextSplitter들을 활용할 수 있고, 교재에서는 **RecursiveCharacterTextSplitter**를 활용했다.

```

chunk_size = 200 # 분할된 청크의 크기를 지정합니다.
chunk_overlap = 20 # 분할된 청크 간의 중복되는 문자 수를 지정합니다.
text_splitter = RecursiveCharacterTextSplitter(
```

```

        chunk_size=chunk_size, chunk_overlap=chunk_overlap
    )

# 문서를 문자 단위로 분할합니다.
splits = text_splitter.split_documents(md_header_splits)
# 분할된 결과를 출력합니다.
for header in splits:
    print(f"{header.page_content}")
    print(f"{header.metadata}", end="n=====\n")

```

TextSplitter.from_language vs. MarkdownHeaderTextSplitter

? 앞서 배운 코드 분할에서 마크다운을 선택하는 것과는 어떤 차이가 있을까

1. Language.MARKDOWN 호출해서 분할하는 방식

- 핵심: `chunk_size` 를 만족시키는 게 우선과제
- 방법: 마크다운에서 자주 등장하는 패턴 (예: `\n\n`, `\n`, 공백 등 + 마크다운 관련 구분자)을 우선순위로 시도하면서 “잘 끊기는 지점”을 찾음
- 결과: `Document(page_content=...)` 만 생기고, “이 청크가 어떤 헤더 아래에 있었는지” 같은 구조 메타데이터는 기본적으로 안 붙는다

| 마크다운을 ‘구조 문서’로 파싱한다기보다, ‘문자열’로 보고 잘 끊기게 `separator`를 추천해주는 방식

2. MarkdownHeaderTextSplitter 호출해서 분할하는 방식

- 핵심: 문서 구조 (섹션/하위 섹션) 보존이 1순위
- 방법: `#`, `##`, `###` 같은 헤더를 기준으로 문서 구조를 이해해서 분할
- 옵션: 헤더 줄을 본문에서 제거/유지(`strip_headers`) 같은 구조적 옵션이 있음

| RAG에서 “섹션 제목”이 검색 품질에 중요할 때 효과가 있다.

07. HTML 헤더 텍스트 분할(HTMLHeaderTextSplitter)

| MarkdownHeaderTextSplitter 와 유사하게 HTML 특화된 TextSplitter이다

| 텍스트를 요소 수준에서 분할하고 각 헤더에 대한 메타데이터를 추가하는 “구조 인식” 청크 생성기

- (a) 관련 텍스트를 의미론적으로 (대략적으로) 그룹화
- (b) 문서 구조에 인코딩된 컨텍스트 풍부한 정보를 보존

하지만 HTML 구조에 따라 의미론적으로 유의미하게 나오는 분할도 있고 유의미하지 않게 나오는 분할도 존재 한다.

```

headers_to_split_on = [
    ("h1", "Header 1"), # 분할할 헤더 태그와 해당 헤더의 이름을 지정합니다.
    ("h2", "Header 2"),
    ("h3", "Header 3"),
]

# 지정된 헤더를 기준으로 HTML 텍스트를 분할하는 HTMLHeaderTextSplitter 객체를 생성합니다.
html_splitter = HTMLHeaderTextSplitter(headers_to_split_on=headers_to_split_on)
# HTML 문자열을 분할하여 결과를 html_header_splits 변수에 저장합니다.
html_header_splits = html_splitter.split_text(html_string)
# 분할된 결과를 출력합니다.
for header in html_header_splits:
    print(f"{header.page_content}")
    print(f"{header.metadata}", end="n=====\n")

```

```

Foo
{}
=====
Some intro text about Foo.
Bar main section Bar subsection 1 Bar subsection 2
{'Header 1': 'Foo'}
=====
Some intro text about Bar.
{'Header 1': 'Foo', 'Header 2': 'Bar main section'}
=====
Some text about the first subtopic of Bar.
{'Header 1': 'Foo', 'Header 2': 'Bar main section', 'Header 3': 'Bar subsection 1'}
=====
Some text about the second subtopic of Bar.
{'Header 1': 'Foo', 'Header 2': 'Bar main section', 'Header 3': 'Bar subsection 2'}
=====

Baz
{'Header 1': 'Foo'}
=====
Some text about Baz
{'Header 1': 'Foo', 'Header 2': 'Baz'}
=====
Some concluding text about Foo
{'Header 1': 'Foo'}
=====
```

| **HTMLHeaderTextSplitter**가 활용되는 방식은 크게 두 가지이다.

1. HTML 문자열을 사용하는 경우

```

from langchain_text_splitters import HTMLHeaderTextSplitter

html_string = """
<!DOCTYPE html>
<html>
<body>
```

```

<div>
  <h1>Foo</h1>
  <p>Some intro text about Foo.</p>
  <div>
    <h2>Bar main section</h2>
    <p>Some intro text about Bar.</p>
    <h3>Bar subsection 1</h3>
    <p>Some text about the first subtopic of Bar.</p>
    <h3>Bar subsection 2</h3>
    <p>Some text about the second subtopic of Bar.</p>
  </div>
  <div>
    <h2>Baz</h2>
    <p>Some text about Baz</p>
  </div>
  <br>
  <p>Some concluding text about Foo</p>
</div>
</body>
</html>
"""

headers_to_split_on = [
    ("h1", "Header 1"), # 분할할 헤더 태그와 해당 헤더의 이름을 지정합니다.
    ("h2", "Header 2"),
    ("h3", "Header 3"),
]

# 지정된 헤더를 기준으로 HTML 텍스트를 분할하는 HTMLHeaderTextSplitter 객체를 생성합니다.
html_splitter = HTMLHeaderTextSplitter(headers_to_split_on=headers_to_split_on)
# HTML 문자열을 분할하여 결과를 html_header_splits 변수에 저장합니다.
html_header_splits = html_splitter.split_text(html_string)
# 분할된 결과를 출력합니다.
for header in html_header_splits:
    print(f"{header.page_content}")
    print(f"{header.metadata}", end="\n=====\n")

```

2. 다른 splitter와 파이프라인으로 연결하고, 웹 URL에서 HTML을 로드하는 경우

- web crawling에서는 꽤나 유용해보이지만 앞서 말한 구조별 분할의 한계점 (제대로 분할하지 못한다) 때문에 활용에는 주의가 필요해보인다.
- 또한 동적 웹페이지의 구성의 경우에도 텍스트 추출이 언제 이루어지는가에 따라서 분할 품질이 달라질 것으로 예상된다.

```
from langchain_text_splitters import RecursiveCharacterTextSplitter
```

```

url = "https://plato.stanford.edu/entries/goedel/" # 분할할 텍스트의 URL을 지정합니다.

headers_to_split_on = [ # 분할할 HTML 헤더 태그와 해당 헤더의 이름을 지정합니다.
    ("h1", "Header 1"),
    ("h2", "Header 2"),
    ("h3", "Header 3"),
    ("h4", "Header 4"),
]

# HTML 헤더를 기준으로 텍스트를 분할하는 HTMLHeaderTextSplitter 객체를 생성합니다.
html_splitter = HTMLHeaderTextSplitter(headers_to_split_on=headers_to_split_on)

# URL에서 텍스트를 가져와 HTML 헤더를 기준으로 분할합니다.
html_header_splits = html_splitter.split_text_from_url(url)

chunk_size = 500 # 텍스트를 분할할 청크의 크기를 지정합니다.
chunk_overlap = 30 # 분할된 청크 간의 중복되는 문자 수를 지정합니다.
text_splitter = RecursiveCharacterTextSplitter( # 텍스트를 재귀적으로 분할하는 RecursiveCharacterTextSplitter 객체를 생성합니다.
    chunk_size=chunk_size, chunk_overlap=chunk_overlap
)

# HTML 헤더로 분할된 텍스트를 다시 청크 크기에 맞게 분할합니다.
splits = text_splitter.split_documents(html_header_splits)

# 분할된 텍스트 중 80번째부터 85번째까지의 청크를 출력합니다.
for header in splits[80:85]:
    print(f"{header.page_content}")
    print(f"{header.metadata}", end="\n=====\n")

```

08. 재귀적 JSON 분할(RecursiveJsonSplitter)

| JSON 데이터를 깊이 우선 탐색(depth-first traversal)하여 더 작은 JSON 청크(chunk)를 생성

```

from langchain_text_splitters import RecursiveJsonSplitter

# JSON 데이터를 최대 300 크기의 청크로 분할하는 RecursiveJsonSplitter 객체를 생성합니다.
splitter = RecursiveJsonSplitter(max_chunk_size=300)

```

| 이 분할기는 **중첩된 JSON 객체를 가능한 한 유지**하려고 시도하지만, 청크의 크기를 `min_chunk_size`와 `max_chunk_size` 사이로 유지하기 위해 필요한 경우 객체를 분할

| 값이 중첩된 JSON이 아니라 **매우 큰 문자열**인 경우, **해당 문자열은 분할되지 않는다.**

| 아래에서도 일부 청크는 사이즈가 사전에 정의된 것보다 훨씬 더 크게 나온다.

```
# 청크의 크기를 확인해 봅시다.
print([len(text) for text in texts][:10])

# 더 큰 청크 중 하나를 검토해 보면 리스트 객체가 있는 것을 볼 수 있습니다.
print(texts[1])
[404, 127, 460, 176, 1327, 211, 202, 214, 238, 344]
{"info": {"version": "0.1.0"}, "paths": {""/api/v1/audit-logs": {"get": {"tags": ["audit-logs"], "summary": "Get Audit Logs"}}}}
```

분할하는 기준

1. 텍스트 분할 방식: JSON 값 기준
2. 청크 크기 측정 방식: 문자 수 기준
 - `split_json(json_data=...)`
 - 작은 JSON 조작(파이썬 dict/list 형태)을 얻고 싶을 때
 - 후속 조작/탐색/필터링에 유리
 - `split_text(json_data=...)`
 - 분할 결과를 문자열(JSON string) 리스트로 받고 싶을 때
 - 바로 임베딩/저장소로 넣기 쉬움
 - `create_documents(texts=[json_data])`
 - 분할 결과를 Document 객체로 받음
 - `page_content`에 JSON 문자열이 들어감
- `max_chunk_size`를 줘도 리스트(list)가 포함된 청크는 제한을 초과할 수 있다
- 청크 크기를 엄격히 제한해야 하면, JSON 분할 후에 **Recursive Text Splitter**로 후처리하는 것을 고려한다.