**🔹 1. *Updated Code with Token Chunking***

**Purpose:**

* Specifically designed for **token-based splitting** (tokens, not characters or words).
* Useful when working with **LLMs (like GPT, Gemini)** because APIs have **token limits**.
* Instead of guessing text size by characters, it splits by exact tokens.

**Where to Use:**

* When preparing data for **embedding models** or **LLM calls**.
* For **RAG pipelines** where token control is critical (e.g., max 512 tokens per chunk).
* When you want **precise control over model cost and context length**.

**Advantage:**  
✅ Exact token counts (important for OpenAI, Gemini, Anthropic, etc.).  
✅ Avoids unexpected truncation or over-spending on tokens.  
✅ Perfect for production RAG apps.

✅ token\_chunking() splits text **by tokens** instead of words or characters.  
✅ Uses tiktoken, the same library OpenAI uses internally, so it matches LLM token counts.  
✅ Works fully **offline**, no API key needed.  
✅ You can set chunk\_size in tokens (e.g., 20 tokens per chunk).

**🔹 *2. Full Updated Code with Metadata***

**Purpose:**

* Extended previous code to add **metadata** like:
  + chunk\_text: The text itself
  + start\_index / end\_index: Exact character positions in the source text
  + token\_count: Number of tokens in this chunk (if token chunking is used)

**Where to Use:**

* When you need to **trace chunks back** to the original document.
* For **highlighting search results** or linking back to original positions.
* For **analytics** (like chunk size stats, token usage reports).

**Advantage:**  
✅ Enables **traceability**: Know where each chunk comes from.  
✅ Easy to debug chunking strategy and tune performance.  
✅ Great for document-heavy apps (e.g., PDFs, contracts, research papers).

Enhance the class so that **every chunk method can return metadata** like:

* chunk\_text (the actual text)
* start\_index (start position in the original text)
* end\_index (end position in the original text)
* token\_count (for token-based chunks)

This will make it **RAG-ready** because you’ll know exactly where each chunk came from and how big it is.

This makes it **easy to:**

* Store chunks in a vector database (Pinecone, FAISS, etc.).
* Retrieve their positions in the source document.
* Know exactly how many tokens each chunk has (critical for LLM costs).

**🔹 3. *Updated Code with auto\_chunking()***

**Purpose:**

* A **smart wrapper** that automatically picks a chunking strategy **based on text size**.
* Removes the need for you to manually decide which method to use.
* Good for **general-purpose applications** where documents vary in size.

**Where to Use:**

* If you process **mixed-size documents** (short notes, medium docs, long books).
* When building a **pipeline that should “just work”** without tweaking chunking manually.
* Early prototyping, so you don’t waste time deciding per doc.

**Advantage:**  
✅ Automation: Handles chunking strategy automatically.  
✅ Versatile: Adapts to small/medium/large documents.  
✅ Saves dev time — you call auto\_chunking(), it does the rest.

✅ auto\_chunking() automatically picks the best method:

* Small text → sentence\_chunking()
* Medium text → paragraph\_chunking()
* Longer text → fixed\_size\_chunking() or token\_chunking().

✅ Still returns **metadata** (text, start/end index, token count).  
✅ Easy to use in RAG pipelines — just call auto\_chunking().

**🔹 Quick Comparison Table**

| **Feature** | **Token Chunking** | **Full Code w/ Metadata** | **Auto Chunking** |
| --- | --- | --- | --- |
| **Focus** | Tokens only (for LLMs) | All chunk types + metadata | Smart strategy selection |
| **Granularity** | Token-level precision | Character positions, token counts | Automated |
| **Ideal For** | RAG, LLM APIs | Search, traceability, analytics | General use, mixed doc sizes |
| **Flexibility** | Medium (token-based only) | High (choose manually) | High (chooses automatically) |
| **Best When** | You know token limits & want control | You need detailed debugging, indexing | You want simple “plug & play” |

**🔹 Example Use Cases**

1. **Token Chunking Example**  
   You’re building a **ChatGPT-powered search engine**:
   * Each chunk must be **≤ 512 tokens**.
   * You use token chunking to precisely control cost and avoid truncation.
2. **Full Metadata Example**  
   You’re creating a **legal document analyzer**:
   * Need to **highlight** sections in the original document when retrieved.
   * Metadata gives you exact start/end character positions.
3. **Auto Chunking Example**  
   You’re building a **personal note summarizer**:
   * Some notes are **1 paragraph**, others are **20 pages**.
   * auto\_chunking() picks the right strategy without manual tuning.

**🔹 TL;DR**

* **If you’re prototyping:** Start with auto\_chunking() (easy, flexible).
* **If you’re building for LLM APIs:** Use token\_chunking() (precise, cost-efficient).
* **If you’re building a search/indexing pipeline:** Use **Full Metadata** version to track and debug.