**Progress Report**

MY GitHub for this topic: “<https://github.com/monirmo97/Grammar_Constrained_Decoding>” and https://github.com/monirmo97/LLM

https://www.youtube.com/@AndrejKarpathy

https://www.youtube.com/watch?v=zduSFxRajkE

https://github.com/brevdev/notebooks/blob/main/mistral-finetune-own-data.ipynb

https://medium.com/@thakermadhav/build-your-own-rag-with-mistral-7b-and-langchain-97d0c92fa146

https://www.langchain.com/

<https://huggingface.co/blog/how-to-generate>

Tokenization: <https://github.com/openai/openai-cookbook>

Fine-tuning:

https: //[github.com/brevdev/notebooks/blob/main/mistral-finetune-own-data.ipynb](http://github.com/brevdev/notebooks/blob/main/mistral-finetune-own-data.ipynb)[first link]

Future: <https://github.com/ashishpatel26/LLM-Finetuning> It is advanced for fine-tuning after the first link.

The below link is for Tokenization, especially for LLm:

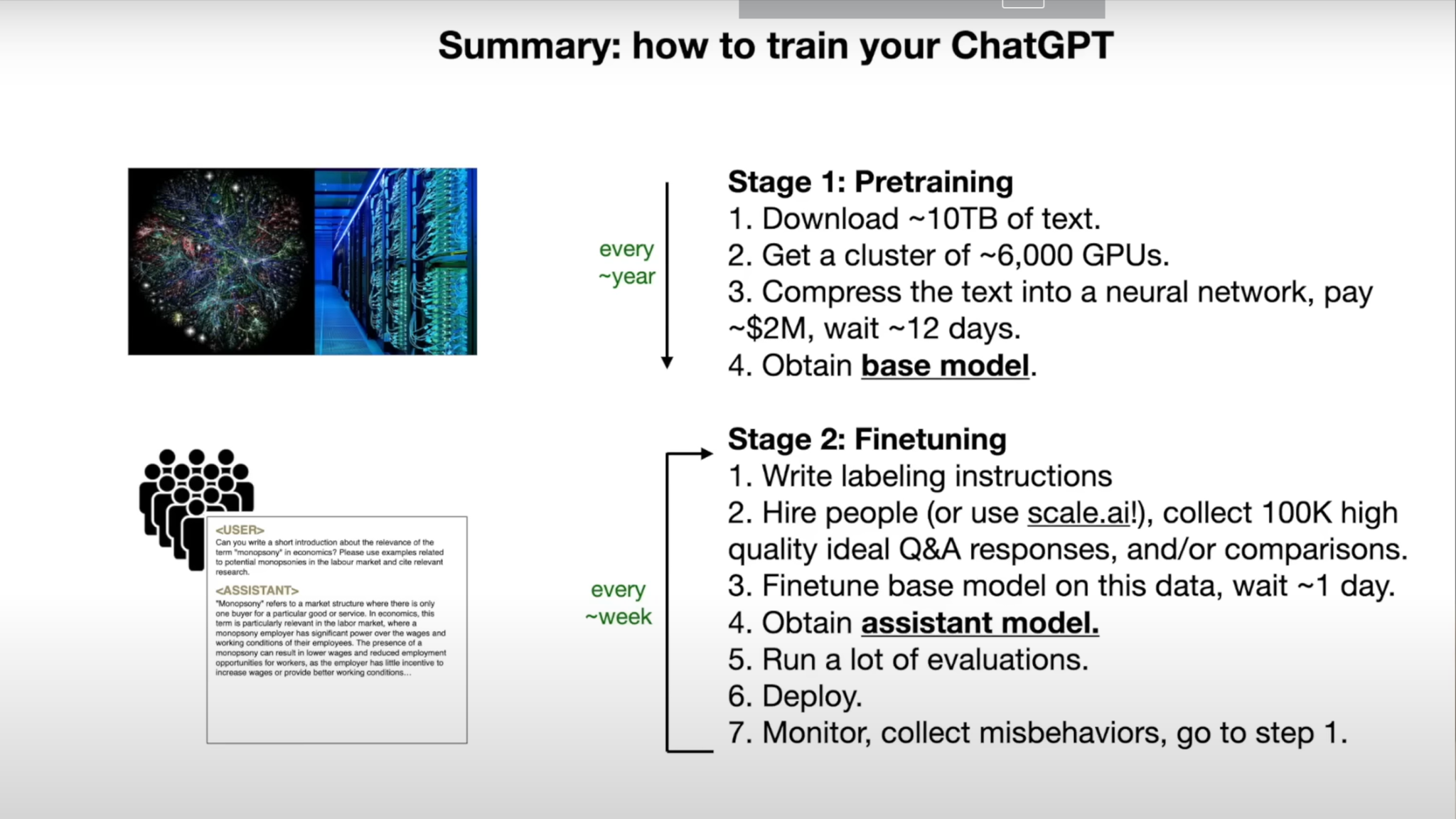
<https://github.com/openai/openai-cookbook/blob/main/examples/How_to_count_tokens_with_tiktoken.ipynb>

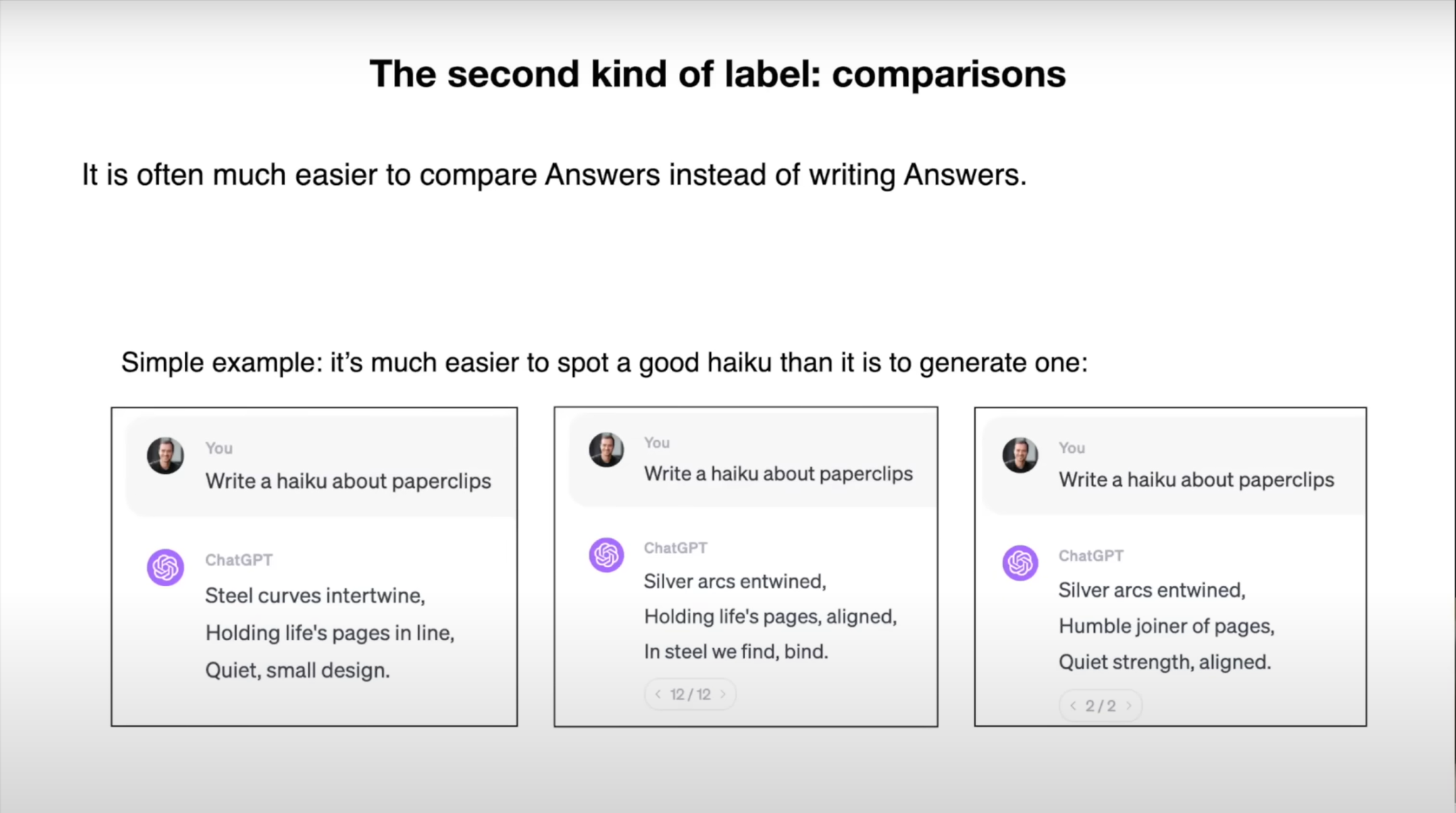
Self-supervised: <https://arxiv.org/pdf/2310.06825>

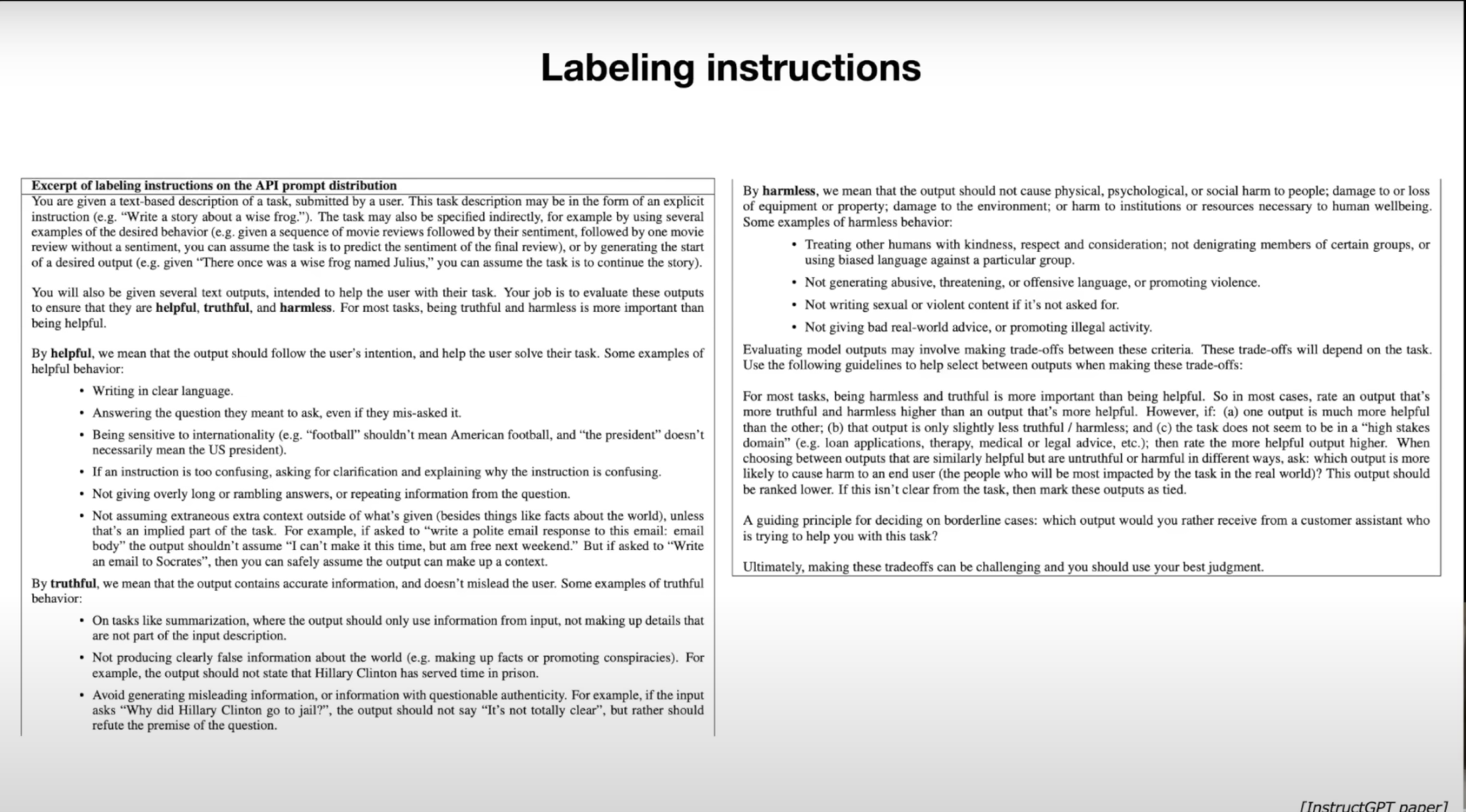
Maybe it is a survey for constraint decoding: <https://arxiv.org/pdf/2403.01632>

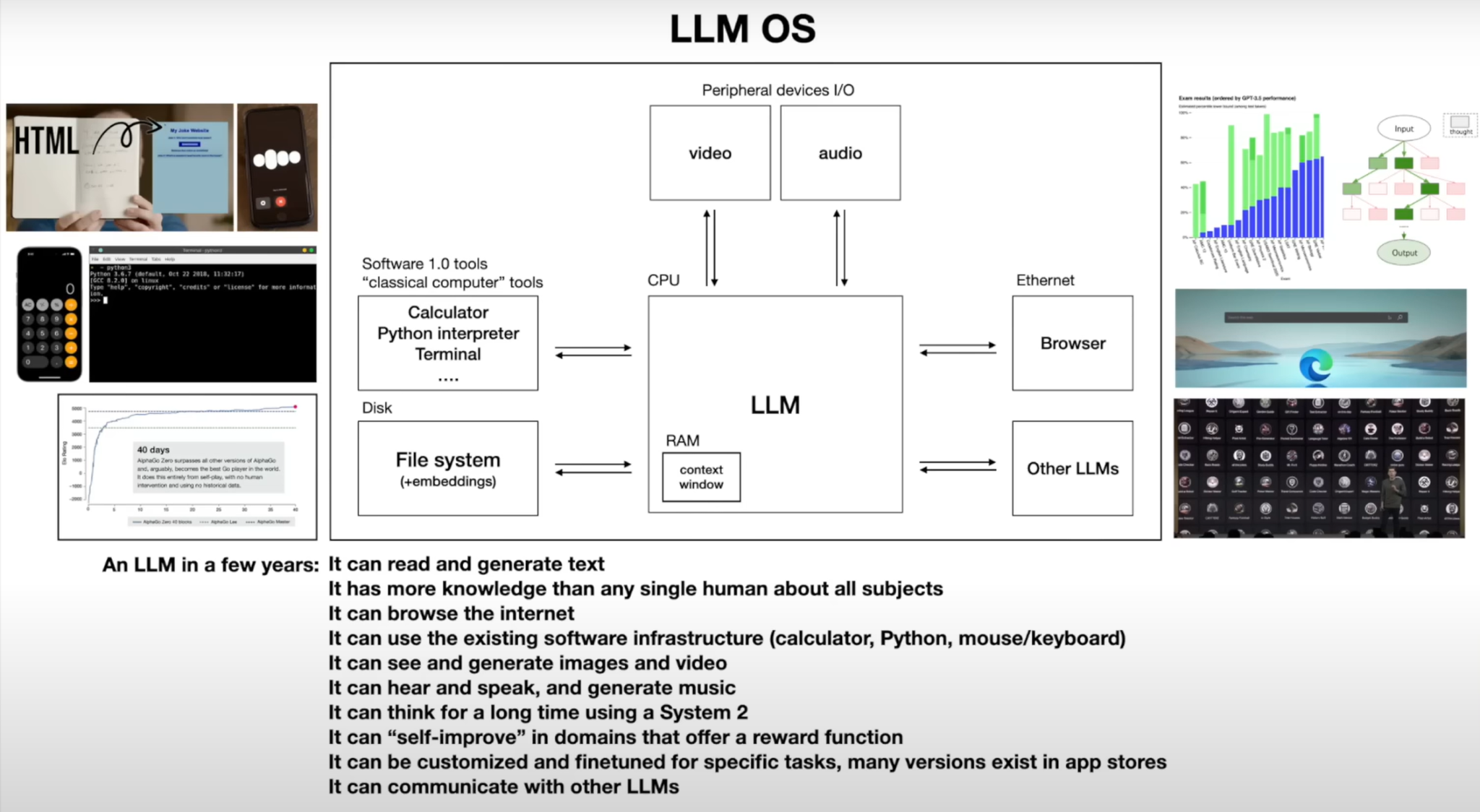
**Papers:**

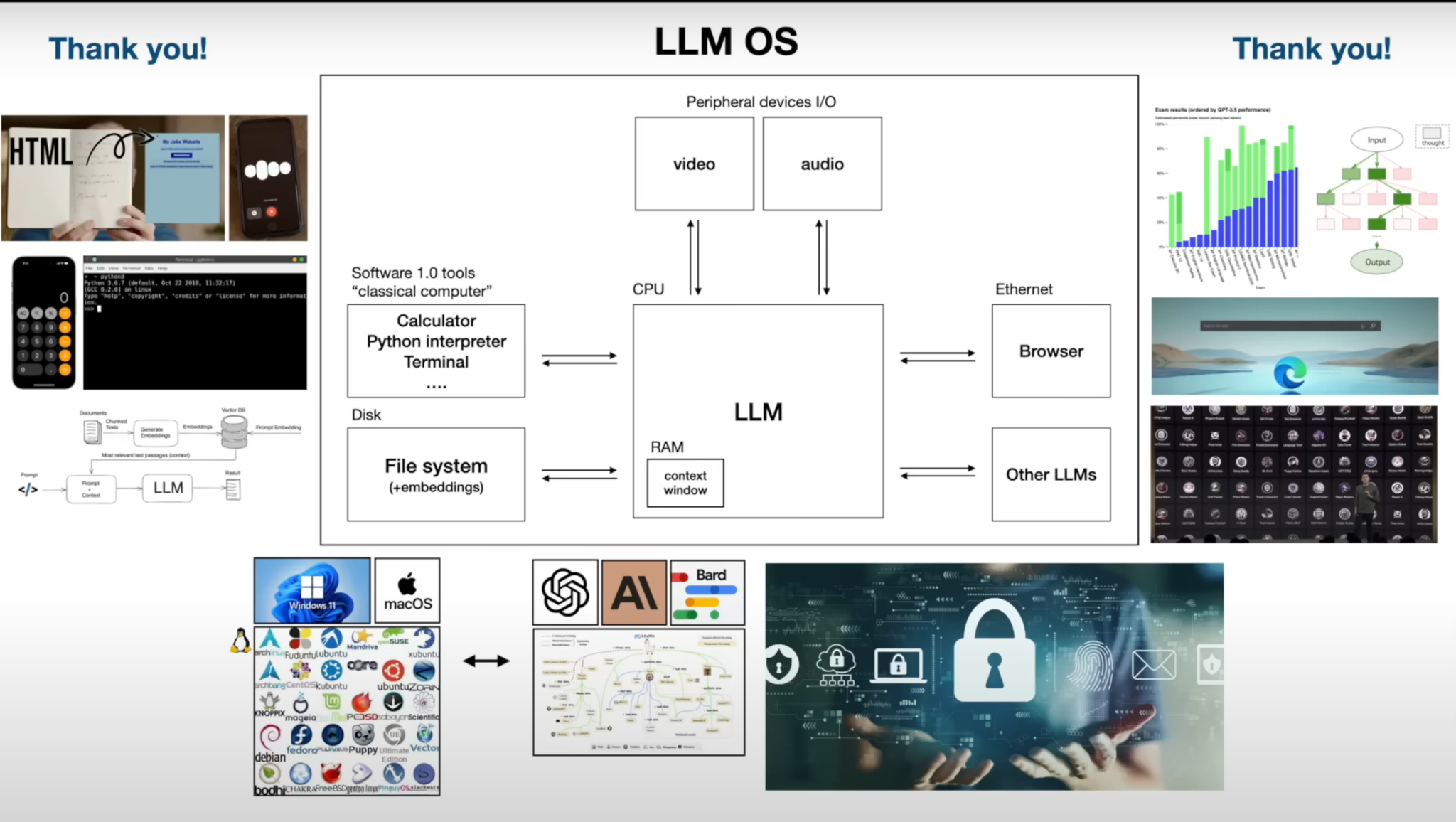
1. Grammar-constrained decoding for Structured NLP Tasks without fine-tuning
2. The Whole Truth and Nothing But the Truth: Faithful and Controllable Dialogue Response Generation with Dataflow Transduction and Constrained Decoding











My insight: This video was a short theory about LLM models. The presenter points to the key points of LLM.

The last part was very interesting for me. It was about attacks in LLm models. I would like to know more about this.

**Task 2.** Read this paper [Grammar-Constrained Decoding for Structured NLP Tasks without Fine Tuning]

**Status:**

### **Introduction to the Paper**

The paper titled **"Grammar-Constrained Decoding for Structured NLP Tasks without Fine Tuning"** explores a method to improve how language models (like GPT-3) generate structured text.

At first, I was required to learn the main concept of this topic:

**What is Structured text?** is the text that follows a specific format or set of rules, such as filling out a form or generating code.

**Structured NLP Tasks:** These tasks require the output to follow a specific format.

**Ex of Structured NLP Tasks:**

* **Information Extraction:** Extracting specific information from text.
* **Entity Disambiguation:** Identifying the correct entity (like a person or place) mentioned in the text.
* **Constituency Parsing:** Analyzing the grammatical structure of a sentence.

**What is the Grammar-Constrained Decoding (GCD)?**

This method ensures that the text the AI generates follows a predefined structure or grammar. Think of it as giving the AI strict rules when generating text.

I need clear and understandable examples of grammar constraint decoding for better understanding.

**Ex of GCD:**

1. Closed Information Extraction (cIE) is a struchtured NLP task

Closed Information Extraction (cIE) involves extracting structured information, specifically subject-relation-object triplets, from unstructured text.

**Task:** Extract subject-relation-object triplets from the text.

**Input Text:** "Marie Curie discovered radium in 1898."

**Desired Output:** [subject: Marie Curie, relation: discovered, object: radium, year: 1898]

**Grammar Constraints:**

* Subjects must be valid entity names.
* Relations must be valid relation names.
* Objects must be valid entity names.
* Years must be valid years.

**Grammar Rules:**

* S -> [subject] [relation] [object] [year]
* [subject] -> "Marie Curie" | "Albert Einstein" | "Isaac Newton"
* [relation] -> "discovered" | "invented" | "proposed"
* [object] -> "radium" | "the theory of relativity" | "calculus"
* [year] -> "in 1898" | "in 1905" | "in 1687"

**Decoding Steps:**

1. **Generate Subject:**
   * Allowed Tokens: "Marie Curie", "Albert Einstein", "Isaac Newton"
   * Model Suggests: "Marie Curie"
   * Output: "Marie Curie"
2. **Generate Relation:**
   * Allowed Tokens: "discovered", "invented", "proposed"
   * Model Suggests: "discovered"
   * Output: "discovered"
3. **Generate Object:**
   * Allowed Tokens: "radium", "the theory of relativity", "calculus"
   * Model Suggests: "radium"
   * Output: "radium"
4. **Generate Year:**
   * Allowed Tokens: "in 1898", "in 1905", "in 1687"
   * Model Suggests: "in 1898"
   * Output: "in 1898"

**Final Output:**

* [subject: Marie Curie, relation: discovered, object: radium, year: 1898]

### **Why is GCD Important?**

* **Current Challenges:** LLMs are good at generating text but struggle with tasks requiring a specific structure without fine-tuning (additional training for specific tasks).
* **Solution with GCD:** Using formal grammar (rules), GCD can guide the AI to produce text that follows the required structure without needing extra training.

### **How does GCD work?**

1. **Formal Grammars:** These are sets of rules that define the structure of the output. For example, in information extraction, the output might need to be in the format "subject-verb-object" (e.g., "John-buys-apple").
2. **Input-Dependent Grammars:** These adjust the rules based on the input text, allowing the AI to generate different structures as needed.

### **What are the Benefits of GCD?**

* **Flexibility:** GCD can be used for various tasks without needing to fine-tune the AI for each task.
* **Performance:** In experiments, GCD-enhanced models performed better than those without constraints, even matching or surpassing some task-specific models.

#### 

#### **Problem:**

LLMs are very good at generating text but struggle when it comes to generating text that follows specific structures or formats without additional training (fine-tuning).

#### **Solution: Grammar-constrained decoding (GCD)**

In this paper, GCD is a method proposed to guide the text generation of LLMs by using formal grammar to ensure the output follows a required structure.

### **Main Contributions of the Paper:**

1. **Unified Framework:** The paper demonstrates that many structured NLP tasks can be framed as grammar-constrained decoding problems.
2. **Input-Dependent Grammars:** Introduces grammars that can adjust based on the input text, allowing for more flexibility in generating different structures for different inputs.
3. **Empirical Demonstration:** Shows through experiments that GCD-enhanced LLMs perform significantly better in tasks like information extraction, entity disambiguation, and constituency parsing

* and…..

**What is tokenization?**

Reference: “<https://huggingface.co/docs/transformers/en/preprocessing>”

Before you can train a model on a dataset, it needs to be preprocessed into the expected model input format. Whether your data is text, images, or audio, they must be converted into tensors batches. In Hugin Face, there is a library (Transformers: it provides a set of preprocessing classes to help prepare your data for the model.)

### **General Tokenization**

Tokenization is the process of breaking down a text into smaller units called tokens. These tokens can be words, subwords, or characters, depending on the tokenization strategy. Tokenization is a crucial step in natural language processing (NLP) as it converts the raw text into a format that can be easily processed by machine learning models.

* Text uses a [Tokenizer](https://huggingface.co/docs/transformers/en/main_classes/tokenizer) to convert text into a sequence of tokens, create a numerical representation of the tokens, and assemble them into tensors. The main tool for preprocessing textual data is a [tokenizer](https://huggingface.co/docs/transformers/en/main_classes/tokenizer). A tokenizer splits text into *tokens* according to a set of rules. The tokens are converted into numbers and then tensors, which become the model inputs. The tokenizer adds any additional inputs required by the model.

**What is the** [**Tokenizer**](https://huggingface.co/docs/transformers/en/main_classes/tokenizer)**?**

The Hugging Face Tokenizer documentation explains how tokenizers prepare inputs for models, including tokenization, converting tokens to IDs, and encoding/decoding sequences. There are two implementations: a full Python version and a faster Rust-based version. Key classes include PreTrainedTokenizer and PreTrainedTokenizerFast, which manage tokenization methods, adding new tokens, handling special tokens, and more. It also details batch encoding, managing token attributes, and configuring tokenization options such as padding, truncation, and special tokens**.**

**Fast tokenizer :**

* Significant Speed-up with Batched Tokenization: When doing batched tokenization, the PreTrainedTokenizerFast class provides a significant speed-up. This is because it leverages the fast implementation in Rust.
* Mapping Between Original String and Token Space: The PreTrainedTokenizerFast class offers methods to map between the original string (characters and words) and the token space

### **Base Class for All Slow Tokenizers:**

The base class for all slow tokenizers is designed to handle the common tasks associated with tokenization and special token management. It provides methods for downloading, caching, and loading pre-trained tokenizers, as well as adding tokens to the vocabulary in a unified way. This abstraction ensures that we don't need to handle the specific vocabulary augmentation methods for various underlying dictionary structures like Byte Pair Encoding (BPE) or SentencePiece.

### **Common Methods:**

#### **Tokenization**

* **Tokenizing**: Splitting strings into tokens.
* **Converting tokens to IDs**: Mapping token strings to integers and vice versa.
* **Encoding/Decoding**: Tokenizing and converting to integers, or converting integers back to tokens and strings.

**Different types of tokenization?**

1. Word Tokenization:

* Splits text into individual words.
* Simple and intuitive.
* Example: "Hello, world!" → ["Hello", ",", "world", "!"]

1. Subword Tokenization:

* Splits text into subwords or morphemes.
* Useful for handling out-of-vocabulary words and reducing vocabulary size.
* Techniques include Byte-Pair Encoding (BPE) and WordPiece.
* Example: "unhappiness" → ["un", "happiness"] or ["un", "##happy", "##ness"]

1. Character Tokenization:

* Splits text into individual characters.
* It is useful for languages with a large number of unique characters.
* Example: "Hello" → ["H", "e", "l", "l", "o"]

1. Sentence Tokenization:

* Splits text into individual sentences.
* Useful for tasks involving sentence-level processing.
* Example: "Hello world. How are you?" → ["Hello world.", "How are you?"]

**Tokenization in LLM?**

**Start with this reference: https://www.youtube.com/watch?v=zduSFxRajkE&t=60s**

Large Language Models (LLMs) like GPT-2, GPT-3, and others use advanced tokenization strategies to efficiently handle large and diverse vocabularies. The most common tokenization techniques used by these models are Byte-Pair Encoding (BPE) and WordPiece. These methods allow the models to process text in a way that balances the trade-off between vocabulary size and handling out-of-vocabulary words.

**Byte-Pair Encoding (BPE):**

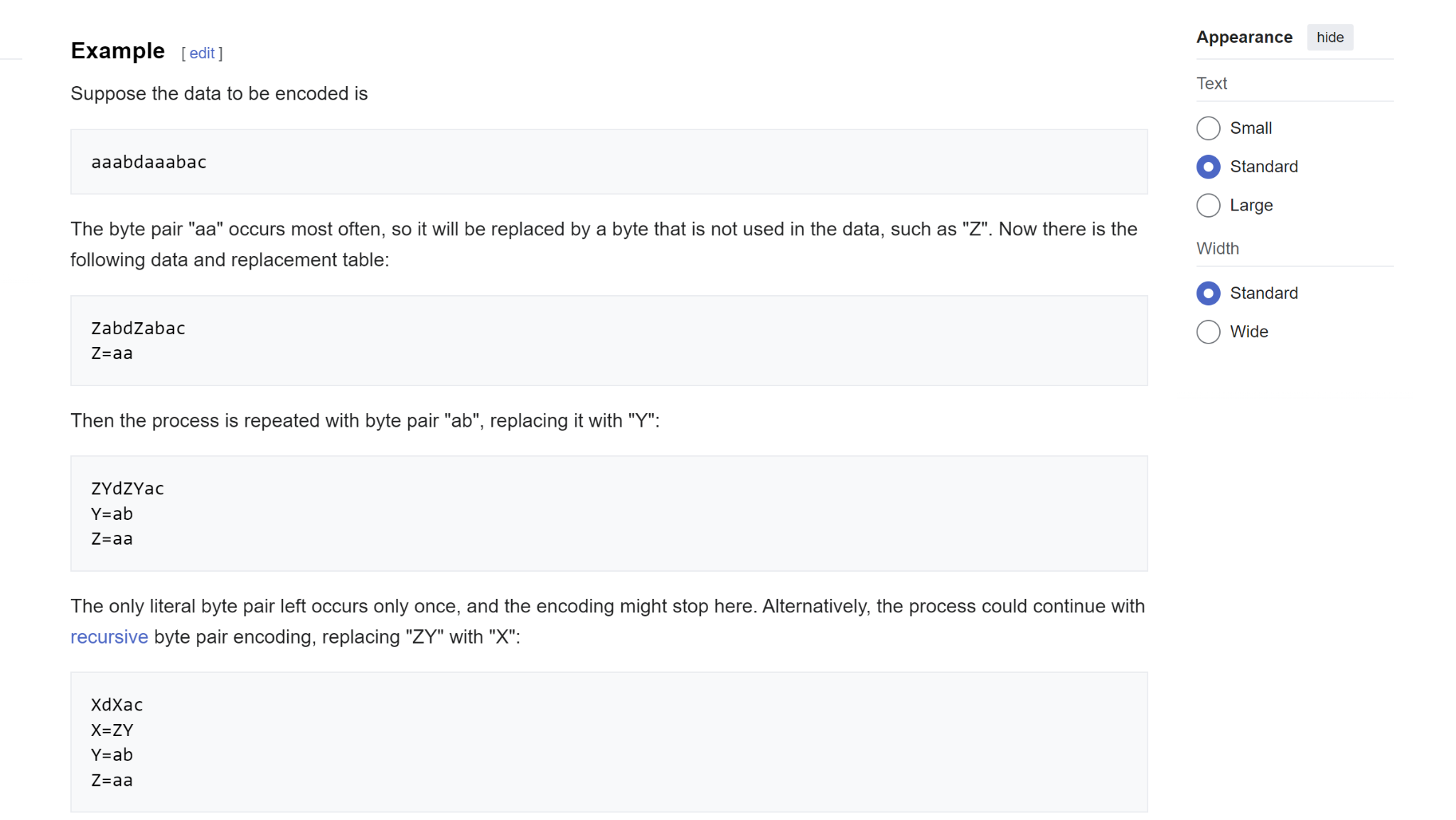
The Byte Pair Encoding (BPE) algorithm is quite instructive for understanding the basic idea of tokenization. Let's walk through an example to see how it works.

Suppose we have a vocabulary of only four elements: a, b, c, and d. Our input sequence is:

***Input sequence: aaabdaaabac***

The sequence is too long, and we'd like to compress it. The BPE algorithm iteratively finds the pair of tokens that occur most frequently and replaces that pair with a single new token.

1. In the first iteration, the byte pair "aa" occurs most often, so it will be replaced by a byte that is not used in the data, such as "Z".
2. The data and replacement table become: ***Zabdaaabac, Z=aa***
3. The process is repeated with byte pair "ab", replacing it with "Y":After the second iteration:***ZYdZYac, Y=ab, Z=aa***
4. In the final round, the pair "ZY" is most common and replaced with "X":
5. After final iteration: ***XdXac, X=ZY, Y=ab, Z=aa***

******

**Result:** After going through this process, instead of having a sequence of 11 tokens with a vocabulary length of 4, we now have a sequence of 5 tokens with a vocabulary length of 4. The BPE algorithm can be applied in the same way to byte sequences. Starting with a vocabulary size of 256, we iteratively find the byte pairs that occur most frequently, mint new tokens, append them to the vocabulary, and replace occurrences in the data. This results in a compressed dataset and an encoding/decoding algorithm.

**implementation of tokenization in large language models (LLMs):**

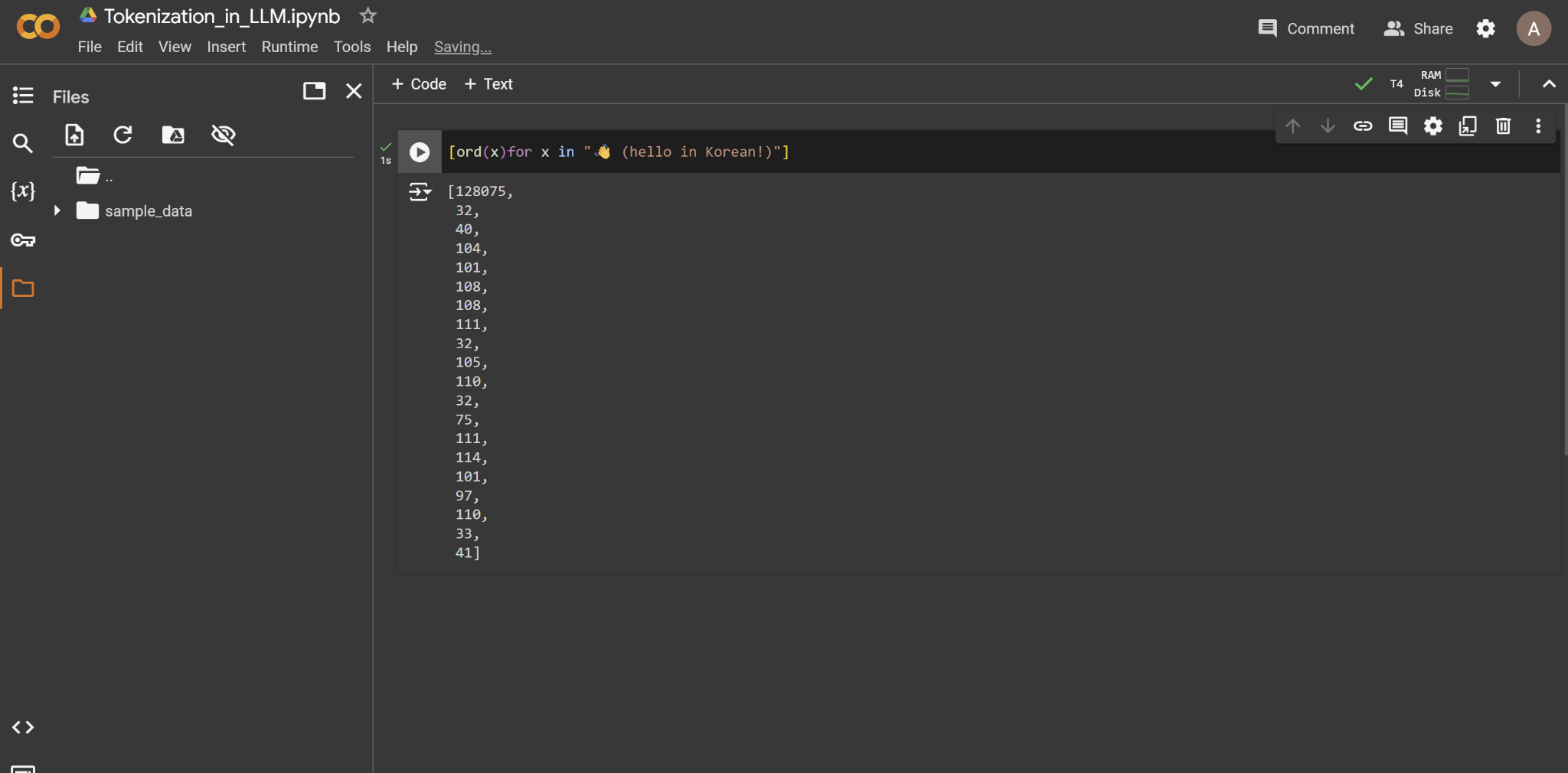
Now, I will start the implementation of tokenization in large language models (LLMs). Tokenization is a crucial component of state-of-the-art LLMs. Still, it is necessary to understand in some detail because a lot of the shining results of LLMs may be attributed to the neural network or otherwise actually traced back to tokenization.

**Unicode tokenization:**

**What is Unicode?**

In Python, strings are immutable sequences of Unicode code points. The Unicode Consortium defines Unicode code points as part of the Unicode standard, which currently defines roughly 150,000 characters across 161 scripts. The standard is alive, with the latest version 15.1 released in September 2023.

We can access the Unicode code point for a single character using Python's ord() function. For example:



*Using ord() to get Unicode code points for characters in a string.*

However, we can't simply use these raw code point integers for tokenization, as the vocabulary would be too large (150,000+) and unstable due to the evolving Unicode standard.

**Unicode Byte Encoding:** To find a better solution for tokenization, we turn to Unicode byte encodings like ASCII, UTF-8, UTF-16, and UTF-32. These encodings define how to translate the abstract Unicode code points into actual bytes that can be stored and transmitted.

The Unicode Consortium defines three types of encodings: UTF-8, UTF-16 and UTF-32. These encodings are how we can take Unicode text and translate it into binary data or byte streams.

### **Encodings: UTF-8, UTF-16, and UTF-32**

To store or transmit these Unicode characters, we need to convert (or encode) them into a sequence of bytes. UTF-8, UTF-16 and UTF-32 are different encoding schemes that specify how these code points are translated into byte streams.

* **UTF-8**: Variable-length encoding (1 to 4 bytes per code point)
* **UTF-16**: Variable-length encoding (2 or 4 bytes per code point)
* **UTF-32**: Fixed-length encoding (always 4 bytes per code point)

### **UTF-8 Encoding**

UTF-8 is the most common encoding scheme because of its efficiency and compatibility with ASCII. It uses a variable number of bytes to encode each character based on its Unicode code point.

#### **How UTF-8 Works**

1. **1 Byte for ASCII**: The first 128 Unicode code points (0 to 127) correspond to ASCII characters and are encoded using a single byte.
2. **2 Bytes for Additional Characters**: The next 1,920 code points (128 to 2,047) are encoded using two bytes.
3. **3 Bytes for BMP**: The following 61,440 code points (2,048 to 65,535) are encoded using three bytes. This covers most of the Basic Multilingual Plane (BMP) characters.
4. **4 Bytes for Supplementary Planes**: Code points above 65,535 (up to 1,114,111) are encoded using four bytes. These include less common characters, such as certain Chinese, Japanese, and Korean (CJK) characters, historic scripts, and mathematical symbols.

#### **Examples**

1. **1-Byte Encoding (ASCII)**:
   * Character: 'A' (U+0041)
   * UTF-8 Encoding: 0x41
2. **2-Byte Encoding**:
   * Character: 'é' (U+00E9)
   * UTF-8 Encoding: 0xC3 0xA9
3. **3-Byte Encoding**:
   * Character: 'ह' (U+0939)
   * UTF-8 Encoding: 0xE0 0xA4 0xB9
4. **4-Byte Encoding**:
   * Character: '😊' (U+1F60A)
   * UTF-8 Encoding: 0xF0 0x9F 0x98 0x8A

### **Detailed Breakdown**

#### **1-Byte Encoding**

For ASCII characters (0 to 127):

* The byte value is simply the ASCII value.
* Example: 'A' (U+0041) is 0x41 in UTF-8.

#### **2-Byte Encoding**

For code points from 128 to 2,047:

* The format is: 110xxxxx 10xxxxxx
* Example: 'é' (U+00E9)
  + Binary: 1110 1001 (U+00E9)
  + UTF-8: 11000011 10101001 (0xC3 0xA9)

#### **3-Byte Encoding**

For code points from 2,048 to 65,535:

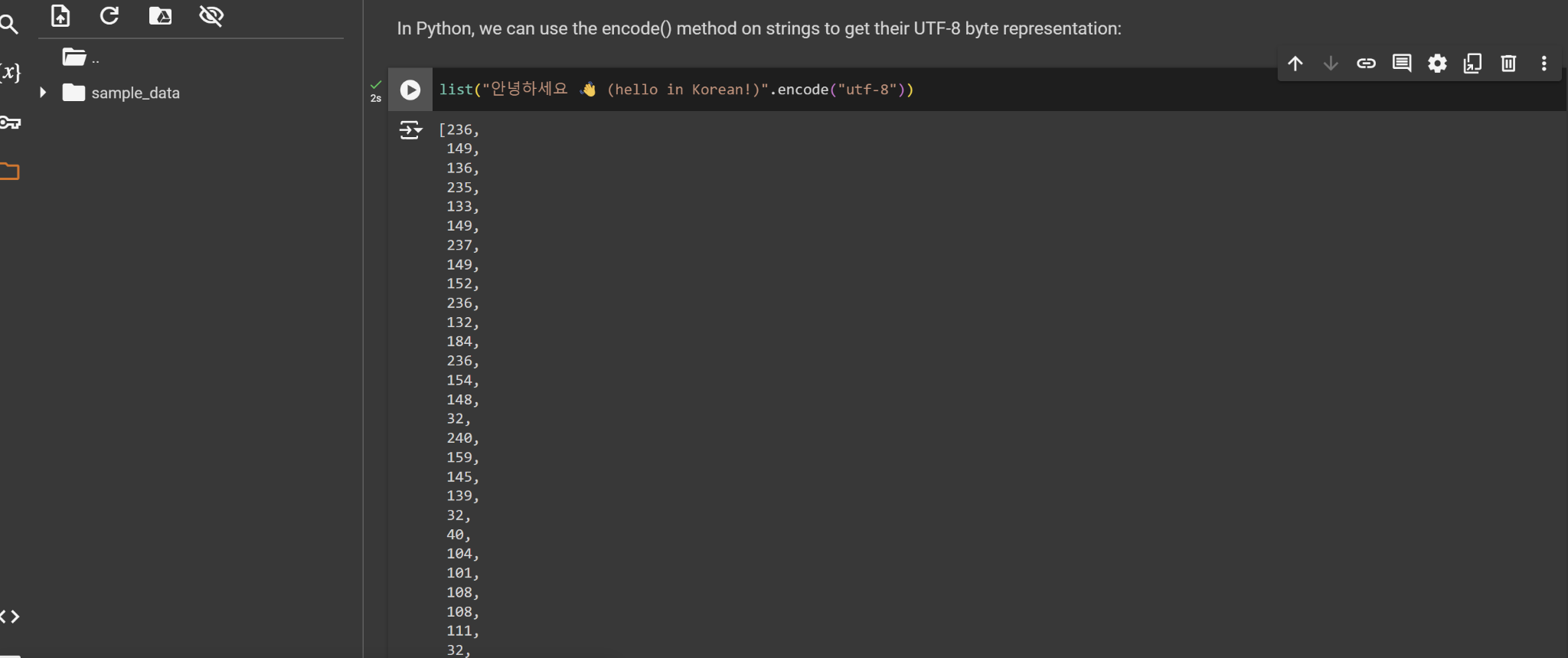
* The format is: 1110xxxx 10xxxxxx 10xxxxxx
* Example: 'ह' (U+0939)
  + Binary: 1001 0011 1001 (U+0939)
  + UTF-8: 11100000 10100100 10111001 (0xE0 0xA4 0xB9)

#### **4-Byte Encoding**

For code points from 65,536 to 1,114,111:

* The format is: 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
* Example: '😊' (U+1F60A)
  + Binary: 0001 1111 0110 0000 1010 (U+1F60A)
  + UTF-8: 11110000 10011111 10011000 10001010 (0xF0 0x9F 0x98 0x8A)

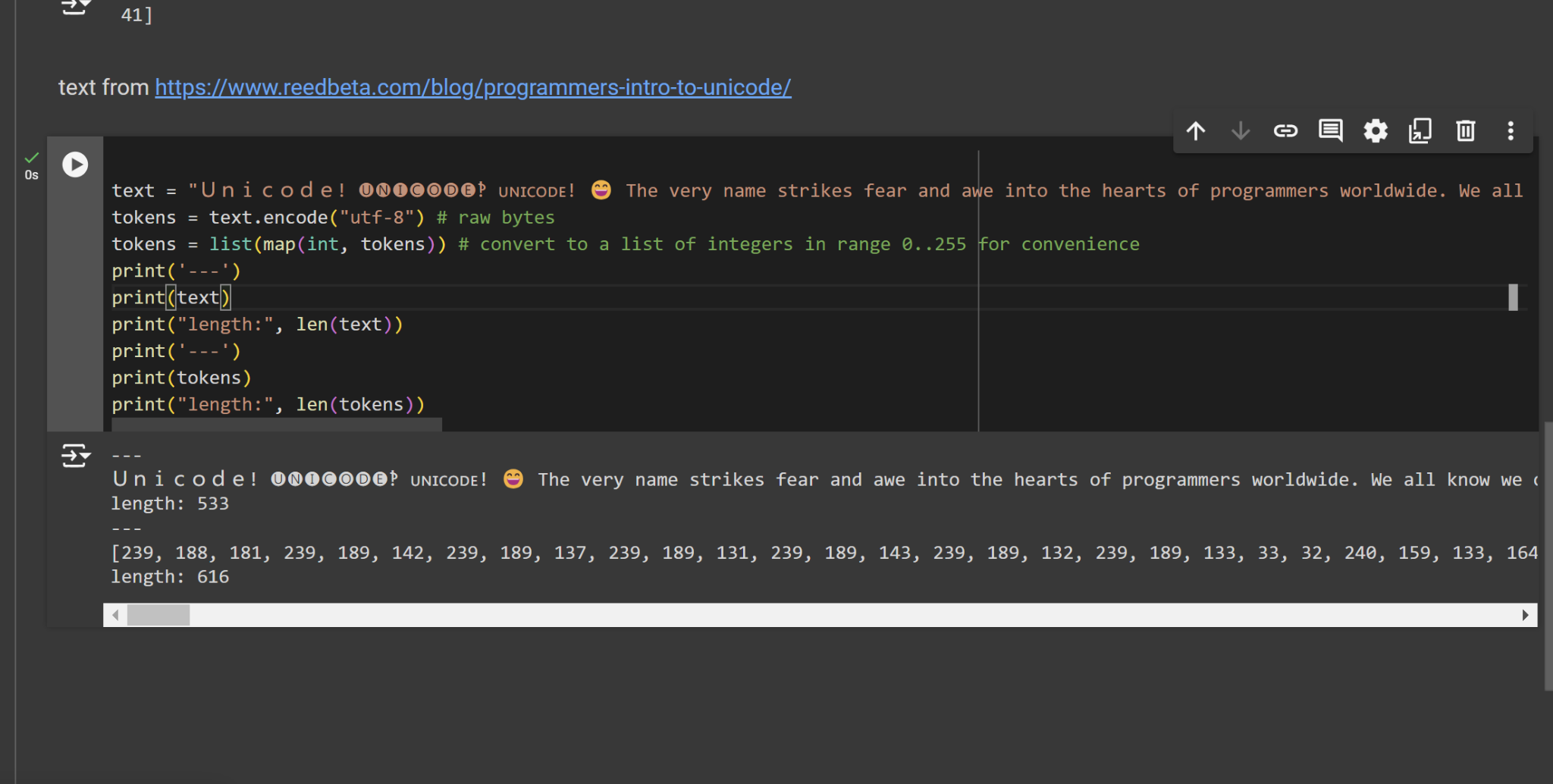
In Python, we can use the encode() method on strings to get their UTF-8 byte representation:



However, directly using the raw UTF-8 bytes would be very inefficient for language models. It would lead to extremely long sequences with a small vocabulary size of only 256 possible byte values. This prevents attending to sufficiently long contexts.

The solution uses a byte pair encoding (BPE) algorithm to compress these sequences to a variable amount. This allows efficient text representation with a larger but tunable vocabulary size.

To get the tokens, we take our input text and encode it into UTF-8. At this point, the tokens will be a raw bytes single stream of bytes. To make it easier to work with, we convert all those bytes to integers and create a list out of it for easier manipulation and visualization in Python.



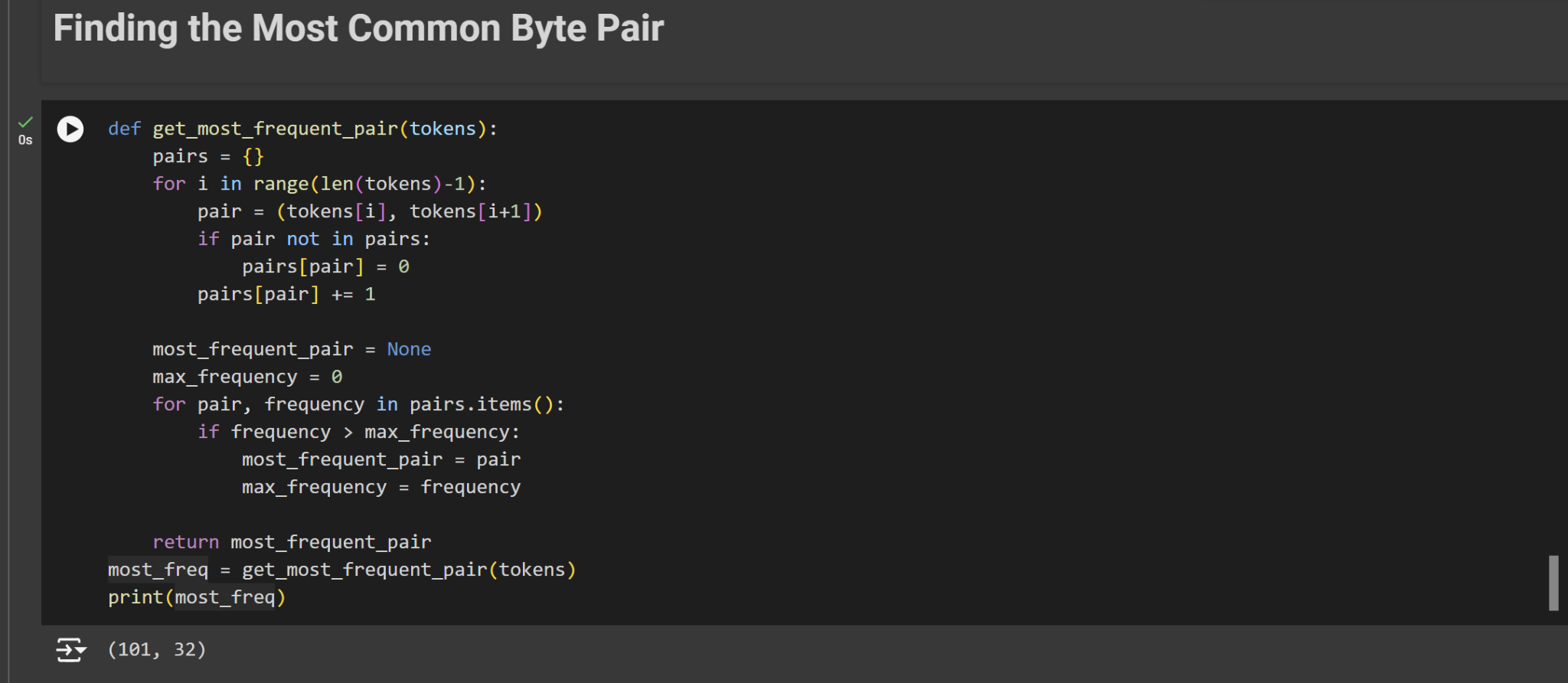
*Converting text to a list of token integers*

The original paragraph has a length of 533 code points, but after encoding into UTF-8, it expands to 616 bytes or tokens. This is because while many simple ASCII characters become a single byte, more complex Unicode characters can take up to four bytes each.

**Finding the most common Byte pair:**

As a first step in the algorithm, we want to iterate over the bytes and find the pair of bytes that occur most frequently, as we will then merge them.

Here is one implementation of **Finding the most common Byte pair** in Python:



This function takes the list of token integers, counts the frequency of each consecutive pair, and returns the pair that appears most often. This is a key step in the byte pair encoding algorithm used for tokenization in many LLMs.

**Finding Most Common Consecutive Pairs in Tokenized Text:**

I will explore how to find the most commonly occurring consecutive pairs in a list of tokenized integers. I'll implement a function called get\_stats that takes a list of integers and returns a dictionary keeping track of the counts of each consecutive pair.

For deeply understanding of tokenization in LLM, please follow the links below:

<https://hundredblocks.github.io/transcription_demo/>

<https://www.youtube.com/watch?v=zduSFxRajkE&t=1468s>

**WordPiece:**

**……..**

**What is the library for tokenization?**

In Python, several libraries support tokenization for natural language processing (NLP). Each library may support different tokenization methods, including word-level, character-level, subword-level (like Byte Pair Encoding and WordPiece), and sentence-level tokenization. Here's a comprehensive list of some of the most commonly used libraries and their supported tokenization methods:

### **1. NLTK (Natural Language Toolkit)**

* **Supported Tokenization Methods**:
  + Word Tokenization: nltk.word\_tokenize
  + Sentence Tokenization: nltk.sent\_tokenize
  + Regular Expression Tokenization: nltk.RegexpTokenizer
  + Character Tokenization (customizable via regex)

**2. spaCy**

* **Supported Tokenization Methods**:
  + Word Tokenization: spacy.tokens.Token
  + Sentence Tokenization: spacy.tokens.Span
  + Custom Tokenization Rules (via the tokenizer attribute)

### **3. Hugging Face Transformers**

* **Supported Tokenization Methods**:
  + WordPiece Tokenization: BertTokenizer
  + Byte Pair Encoding (BPE): GPT2Tokenizer, RobertaTokenizer
  + SentencePiece Tokenization: AlbertTokenizer, T5Tokenizer, XLMTokenizer
  + Unigram Language Model: SentencePieceUnigramTokenizer

### **4. SentencePiece**

* **Supported Tokenization Methods**:
  + Byte Pair Encoding (BPE): sentencepiece.SentencePieceProcessor
  + Unigram Language Model: sentencepiece.SentencePieceProcessor

### **5. Tokenizers (by Hugging Face)**

* **Supported Tokenization Methods**:
  + WordPiece: tokenizers.BertWordPieceTokenizer
  + Byte Pair Encoding (BPE): tokenizers.ByteLevelBPETokenizer
  + SentencePiece: tokenizers.SentencePieceBPETokenizer
  + Unigram: tokenizers.Unigram

### **6. Gensim**

* **Website**: Gensim
* **Supported Tokenization Methods**:
  + Simple Preprocessing: gensim.utils.simple\_preprocess
  + Word Tokenization: gensim.utils.tokenize

**installation steps for each of the mentioned libraries in Google Colab:**

### **1. NLTK (Natural Language Toolkit)**

**!pip install nltk**

### **2. spaCy**

**!pip install spacy**

**# Download the English model**

**!python -m spacy download en\_core\_web\_sm**

### **3. Hugging Face Transformers**

**!pip install transformers**

### **4. SentencePiece**

**!pip install sentencepiece**

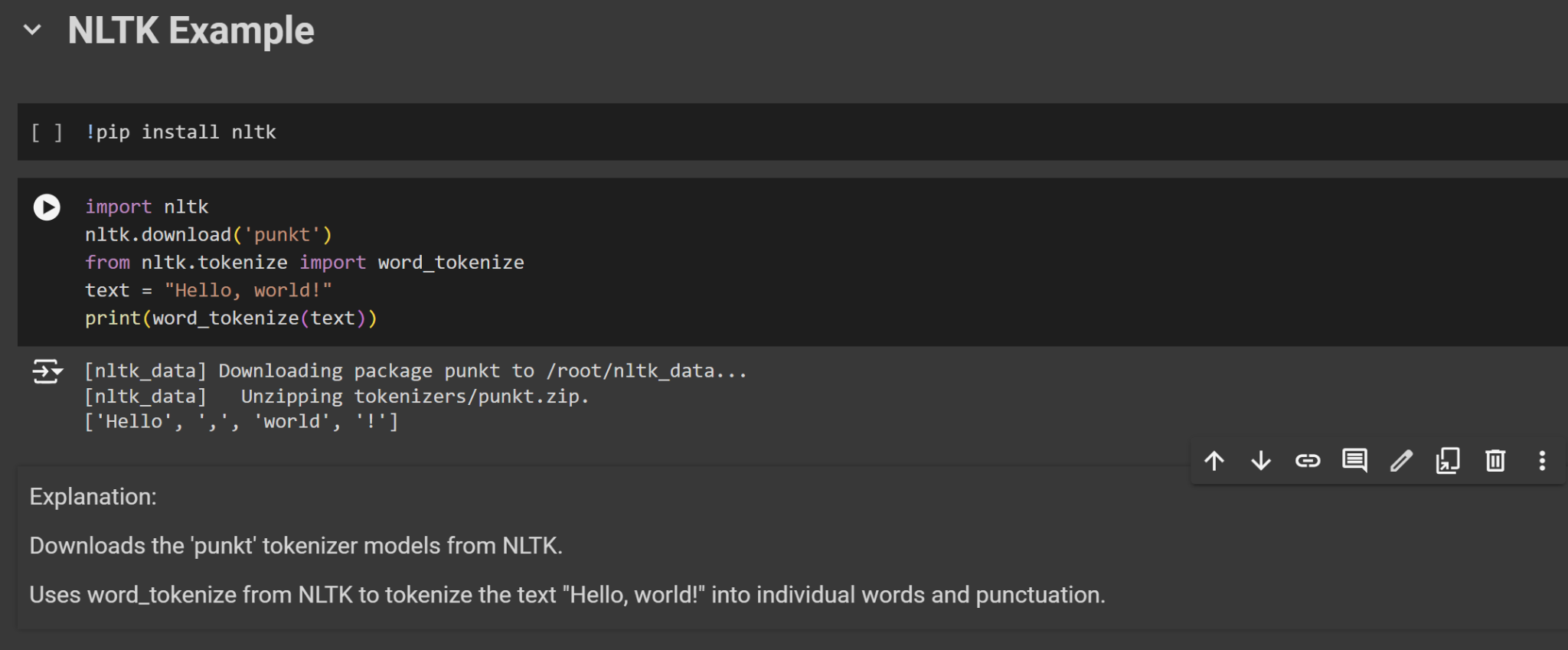
### **5. Tokenizers (by Hugging Face)**

**!pip install tokenizers**

### **6. Gensim**

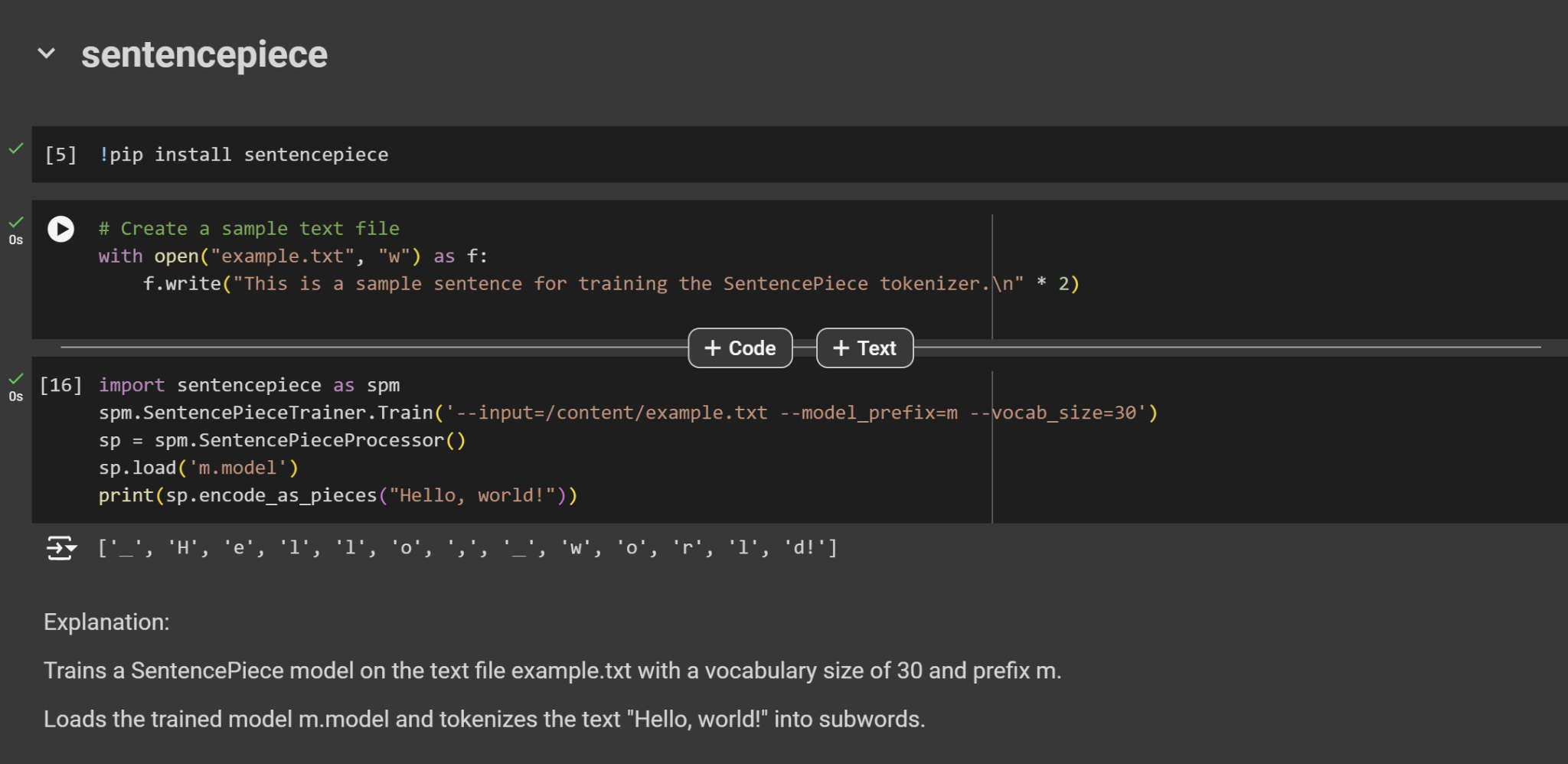
**!pip install gensim**

**Example of tokenization:**

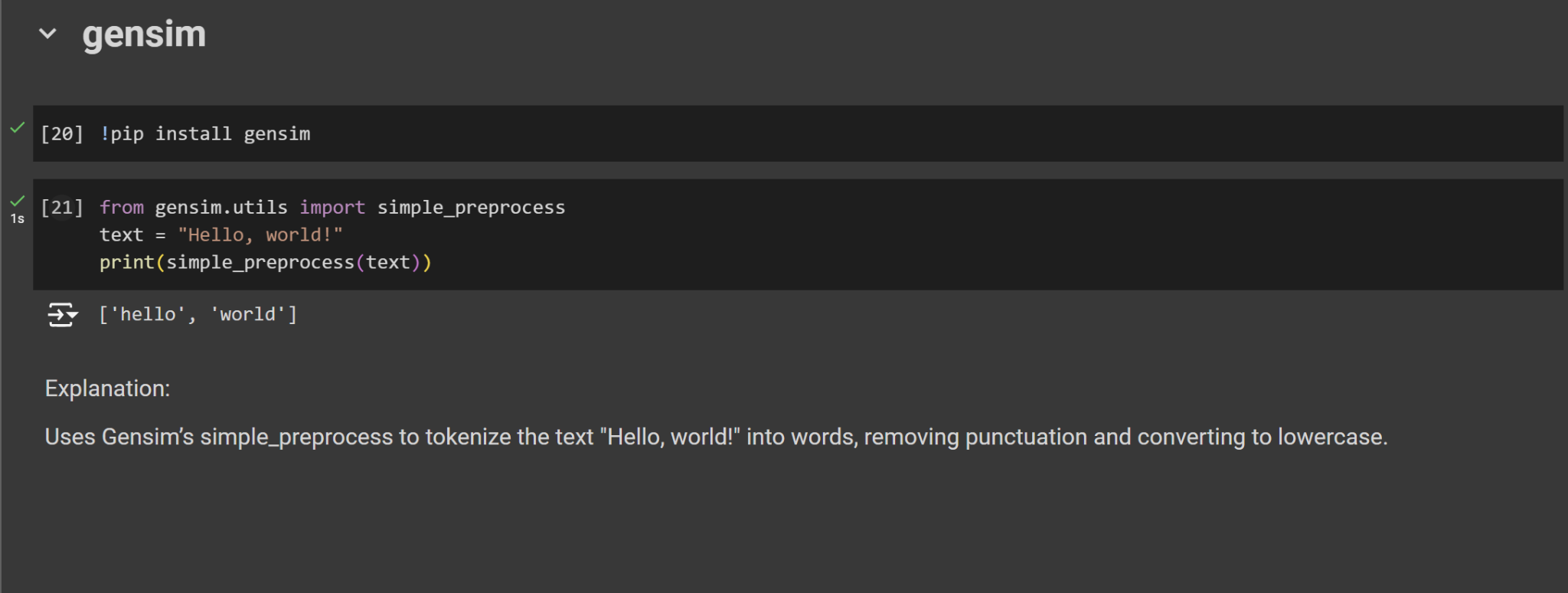
****

****

****

****

****

****

**Methods for Working with LLMs:**

1. Fine-tuning: Involves training the model further on a specific dataset to adapt it to a particular task or domain.

**Example:** Fine-tuning GPT-2 on a wikitext to improve its performance in forecasting the next token. Link to the project I did for this:

<https://github.com/monirmo97/LLM/tree/main/Train_Test_GPT2>

1. Prompt

Of all the inputs to a large language model, by far the most influential is the text prompt. Large language models can be prompted to produce output in a few ways:

* **Instruction:** Tell the model what you want
* **Completion:** Induce the model to complete the beginning of what you want
* **Scenario:** Give the model a situation to play out
* **Demonstration:** Show the model what you want, with either:
  + A few examples in the prompt
  + Many hundreds or thousands of examples in a fine-tuning training dataset

An example of each is shown below.

### **Instruction prompts:** Write your instruction at the top of the prompt (or at the bottom, or both), and the model will do its best to follow the instructions and then stop. Instructions can be detailed, so feel free to write a paragraph explicitly detailing the output you want, just stay aware of how many tokens the model can process.

**Example of instruction prompt:**

Extract the name of the author from the quotation below

“Some humans theorize that intelligent species go extinct before they can expand into outer space. If they're correct, then the night sky's hush is the graveyard's silence.”

Ted Chiang, Exhalation

Output: Ted Chiang

### **Completion prompt:** Completion-style prompts take advantage of how large language models try to write text they think will likely come next. To steer the model, try beginning a pattern or sentence that will be completed by the output you want to see. This mode of steering large language models can take more care and experimentation relative to direct instructions. In addition, the models won't know where to stop, so you will often need stop sequences or post-processing to cut off text generated beyond the desired output.

**Example of completion prompt:**

“Some humans theorize that intelligent species go extinct before they can expand into outer space. If they're correct, then the night sky's hush is the graveyard's silence.”

Ted Chiang, Exhalation

The author of this quote is

Output: Ted Chiang

### **Scenario prompt:** Giving the model a scenario to follow or a role to play out can be helpful for complex queries or when seeking imaginative responses. When using a hypothetical prompt, you set up a situation, problem, or story, and then ask the model to respond as if it were a character in that scenario or an expert on the topic.

**Example of scenario prompt:**

Your role is to extract the name of the author from any given text

“Some humans theorize that intelligent species go extinct before they can expand into outer space. If they're correct, then the night sky's hush is the graveyard's silence.”

Ted Chiang, Exhalation

Output: Ted Chiang

### **Demonstration prompt (few-shot learning):** Like completion-style prompts, demonstrations can show the model what you want it to do. This approach is sometimes called few-shot learning, as the model learns from a few examples in the prompt.

**Example of demonstration prompt:**

Quote:

“When the reasoning mind is forced to confront the impossible repeatedly, it has no choice but to adapt.”

N.K. Jemisin, The Fifth Season

Author: N.K. Jemisin

Quote:

“Some humans theorize that intelligent species go extinct before they can expand into outer space. If they're correct, then the night sky's hush is the graveyard's silence.”

Ted Chiang, Exhalation

Author:

Output: Ted Chiang

### **Fine-tuned prompt:** With enough training examples, you can fine-tune a custom model. In this case, instructions become unnecessary, as the model can learn the task from the training data provided. However, it can be helpful to include separator sequences (e.g., -> or ### or any string that doesn't commonly appear in your inputs) to tell the model when the prompt has ended and the output should begin. With separator sequences, the model can continue elaborating on the input text rather than starting on the answer you want to see.

**Example of fine-tuned prompt:** (for a model that has been custom-trained on similar prompt-completion pairs):

“Some humans theorize that intelligent species go extinct before they can expand into outer space. If they're correct, then the night sky's hush is the graveyard's silence.”

Ted Chiang, Exhalation

Output: Ted Chiang

In addition to text prompts, there are other ways to input data into a large language model:

1. **Multimodal Inputs**: Combining text with images or other types of media (e.g., CLIP models by OpenAI).
2. **Structured Data**: Feeding in tables, graphs, or JSON objects.
3. **Interactive Systems**: Integrating with applications that provide dynamic, real-time inputs, such as chatbots or virtual assistants.
4. **Sensor Data**: Using IoT devices to send real-time sensor data as inputs.
5. **Voice Input**: Converting speech to text using speech recognition systems.

References[<https://github.com/openai/openai-cookbook>]

**The Summary of Paper 1:**

**Theory:**

#### **1. Large Language Models (LLMs):**

* **What are LLMs?** Large language models are powerful AI models trained to understand and generate human language. Examples include GPT-3, BERT, etc.
* **Problem:** LLMs can generate text, but they often struggle with tasks that require structured outputs (e.g., formatted data) without being fine-tuned for specific tasks.

#### **2. Fine-Tuning:**

* **What is Fine-Tuning?** Fine-tuning involves taking a pre-trained model and training it further on a specific task with task-specific data.
* **Drawback:** Fine-tuning can be expensive and requires a lot of task-specific data.

#### **3. Grammar-Constrained Decoding (GCD):**

* **What is GCD?** GCD is a method where predefined grammatical rules guide the language model's text generation. This ensures that the output follows a specific structure without needing fine-tuning.

#### **4. Formal Grammars:**

* **What are Formal Grammars?** These are sets of rules that define valid structures for language. For example, a simple grammar rule might be that a sentence must contain a noun followed by a verb.

### **Main Ideas with Examples**

#### **1. Grammar-Constrained Decoding (GCD):**

* **Idea:** Use grammar rules to control the language model's output.
* **Example:** To generate a valid email address, the grammar ensures the output follows the format "username@domain.com".

#### **2. Input-Dependent Grammars:**

* **Idea:** Grammar rules can change based on the input text.
* **Example:** If the input is "The capital of France is", the grammar ensures the model only considers valid city names (like "Paris").

### **Example Tasks and How GCD Helps**

#### **Task 1: Closed Information Extraction (cIE)**

* **Goal:** Extract subject-relation-object triplets from text.
* **Example:** From "Paris is the capital of France", extract ("Paris", "capital of", "France").
* **Grammar Rule:** Output must follow [subject] [relation] [object].
* **GCD Ensures Output:** "[s] Paris [r] capital of [o] France".

#### **Task 2: Entity Disambiguation (ED)**

* **Goal:** Identify the correct entity from a list of candidates.
* **Example:** For "Apple released a new product," decide if "Apple" refers to the fruit or the tech company.
* **Grammar Rule:** Output must be one of the predefined candidates.
* **GCD Ensures Output:** "Apple (company)".

#### **Task 3: Constituency Parsing (CP)**

* **Goal:** Parse sentences into syntactic trees that represent grammatical relationships.
* **Example:** For "The cat sat on the mat," produce a tree structure showing the grammatical components.
* **Grammar Rule:** Output must be a valid syntactic tree.
* **GCD Ensures Output:** A valid tree structure like "(S (NP The cat) (VP sat (PP on (NP the mat))))".

### **Detailed Concepts with Examples**

#### **1. Grammar-Constrained Decoding (GCD)**

* **How It Works:**
  + **Define Grammar Rules:** Set rules for valid structures.
  + **Example Rule:** For emails, "^[a-zA-Z0-9.\_%+-]+@[a-zA-Z0-9.-]+.[a-zA-Z]{2,}$".
  + **Guide LLM Output:** Use these rules during generation. If the model tries to generate "user@name@domain", it's blocked because it doesn't fit the rule.

#### **2. Input-Dependent Grammars**

* **How It Works:**
  + **Adapt to Input:** Change grammar rules based on input context.
  + **Example:** For the input "The capital of France is", restrict the output to valid city names like "Paris" rather than random words.

### **How GCD Works in Detail:**

1. **Define the Grammar:**
   * Create rules that describe valid output structures.
   * **Example Rule:** For structured data like emails, the rule could be "^[a-zA-Z0-9.\_%+-]+@[a-zA-Z0-9.-]+.[a-zA-Z]{2,}$".
2. **Integrate with LLM:**
   * Use an incremental parser (a tool that checks each step) to guide the LLM.
   * If a token breaks a rule, it's rejected, and the model tries another.
3. **Generate Valid Output:**
   * The model generates text step-by-step, each step checked against the grammar.
   * **Example Workflow:** Generating an email address:
     + **Step 1:** Generate "u".
     + **Step 2:** Generate "s".
     + **Step 3:** Generate "e".
     + **Step 4:** Generate "r".
     + **Step 5:** Generate "@" (all valid so far).
     + **Step 6:** Generate "name@" (invalid — rejected).
     + **Step 6:** Generate "domain.com" (valid — accepted).