**My road map in LLM**

references :

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>

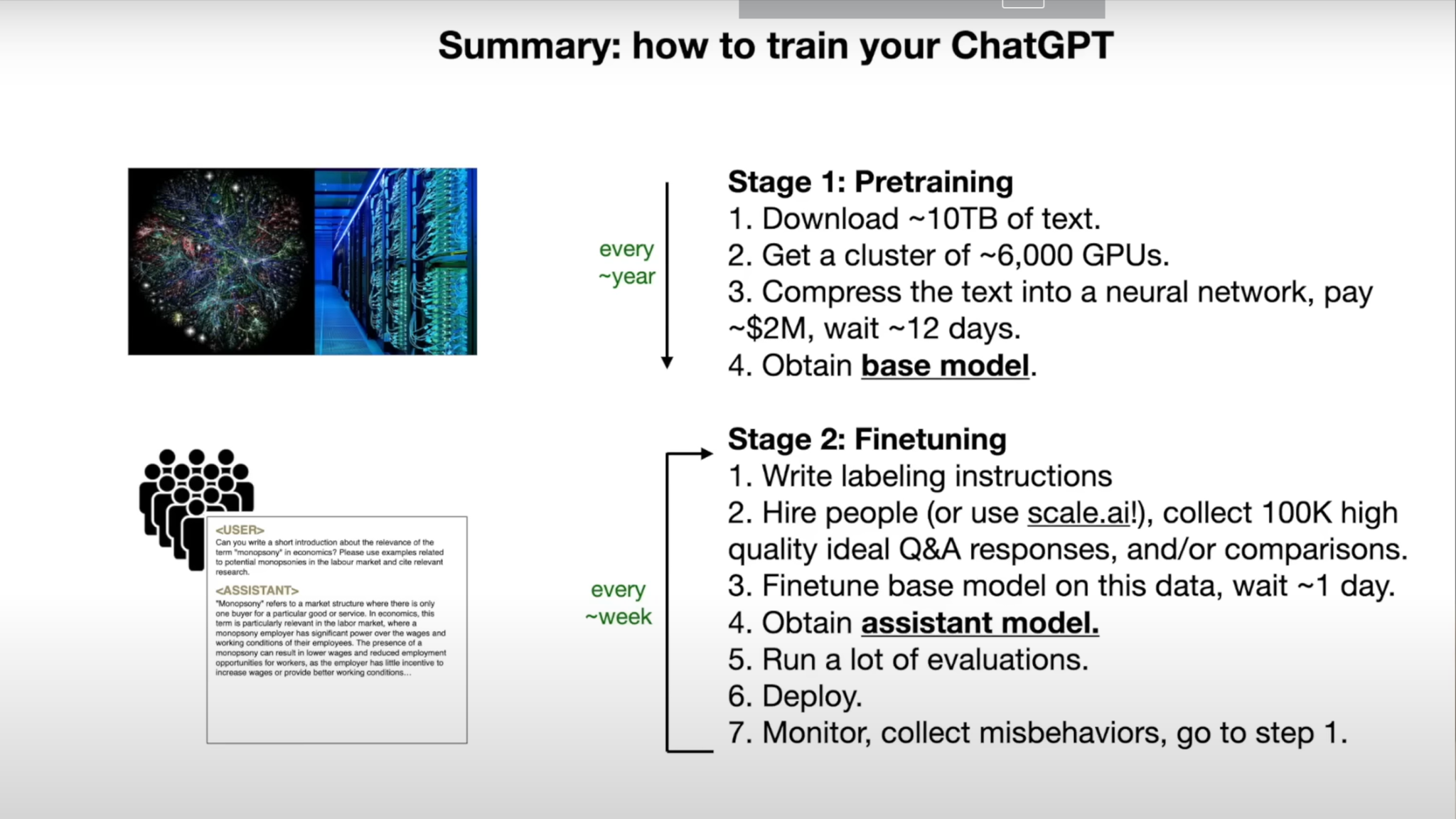
Fine-tuning: https: //[github.com/brevdev/notebooks/blob/main/mistral-finetune-own-data.ipynb](http://github.com/brevdev/notebooks/blob/main/mistral-finetune-own-data.ipynb)[first link]

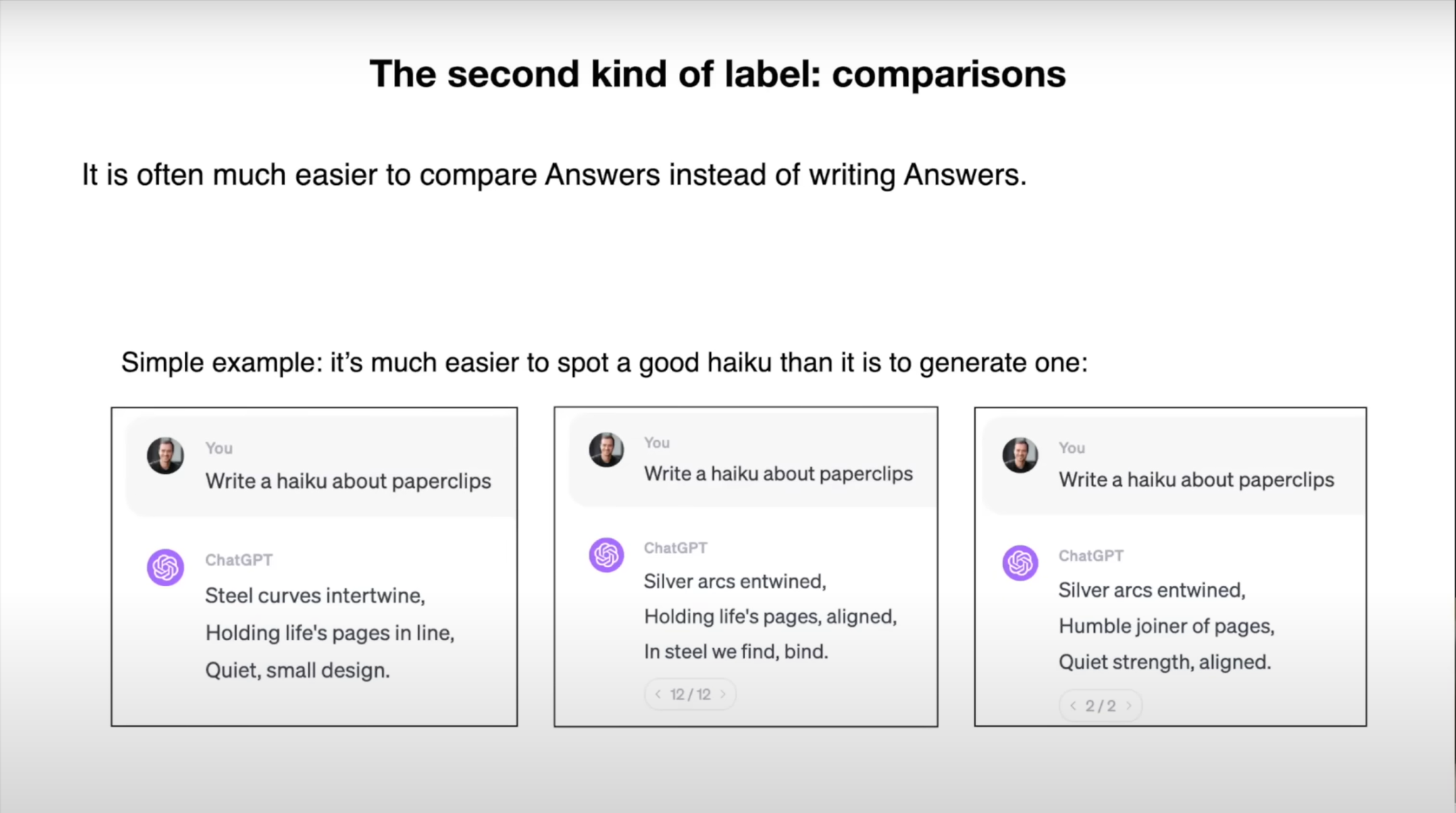
Future: <https://github.com/ashishpatel26/LLM-Finetuning> It is advanced for fine-tuning after the first link

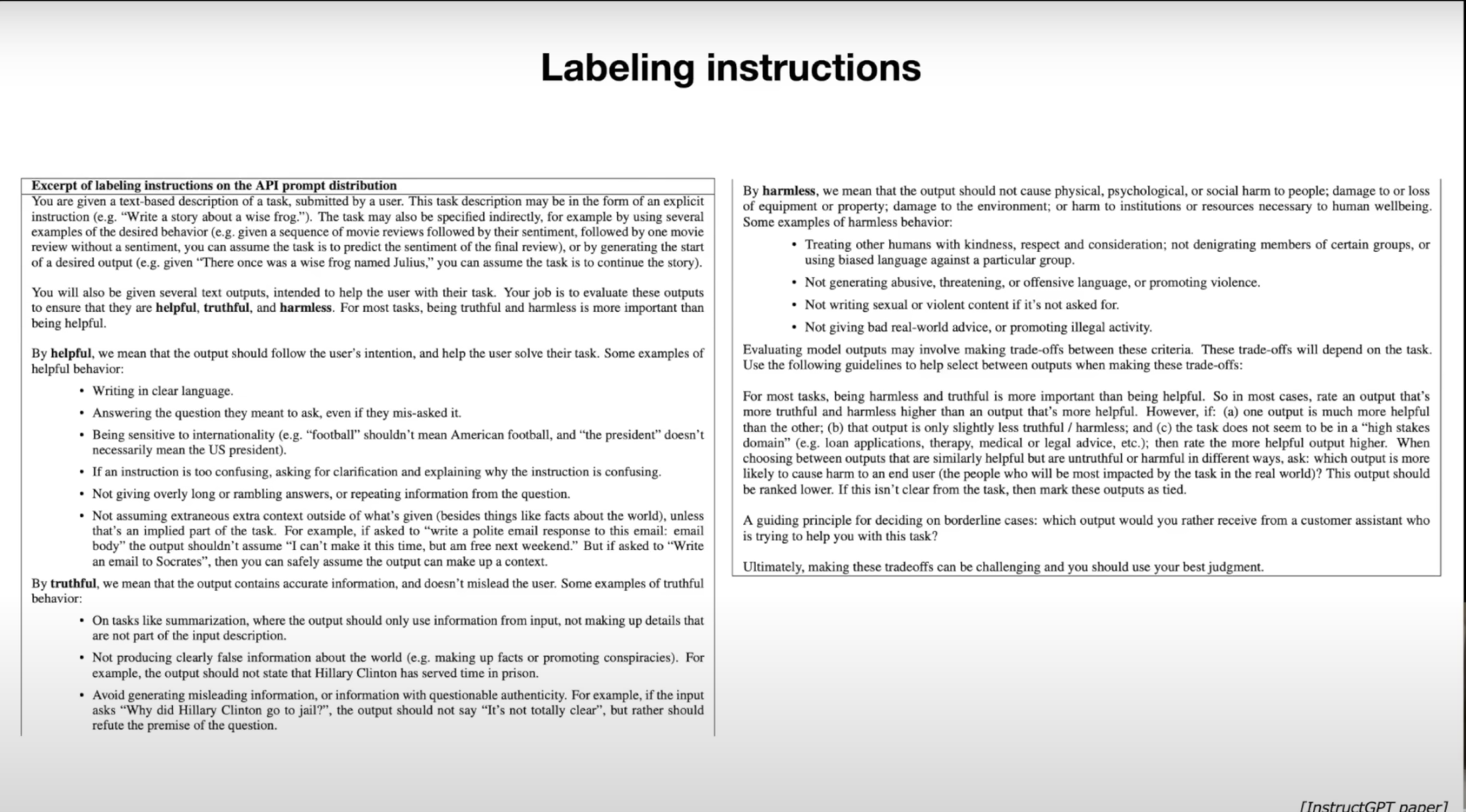
The below link is for Tokenization, especially for LLm:

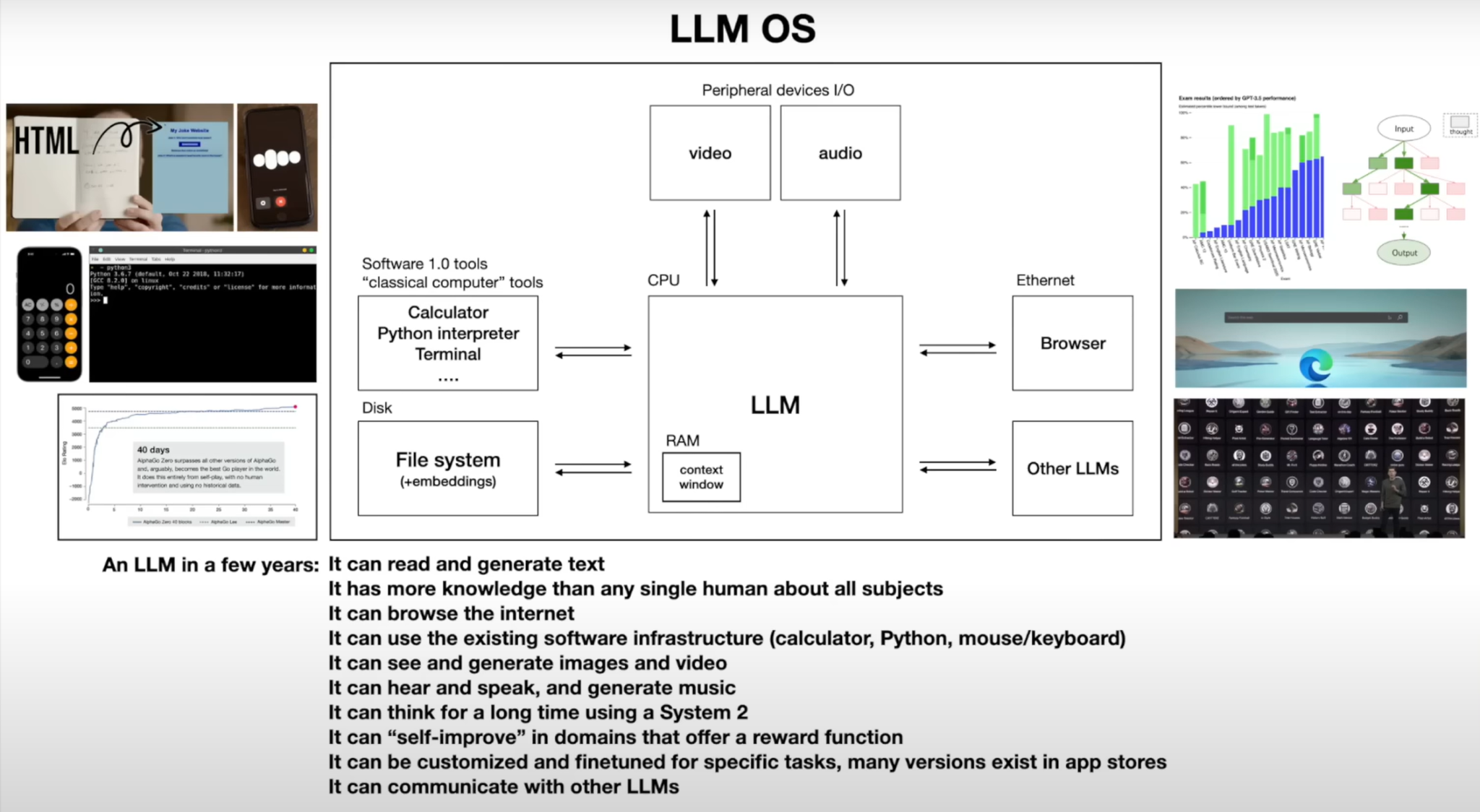
<https://github.com/openai/openai-cookbook/blob/main/examples/How_to_count_tokens_with_tiktoken.ipynb>

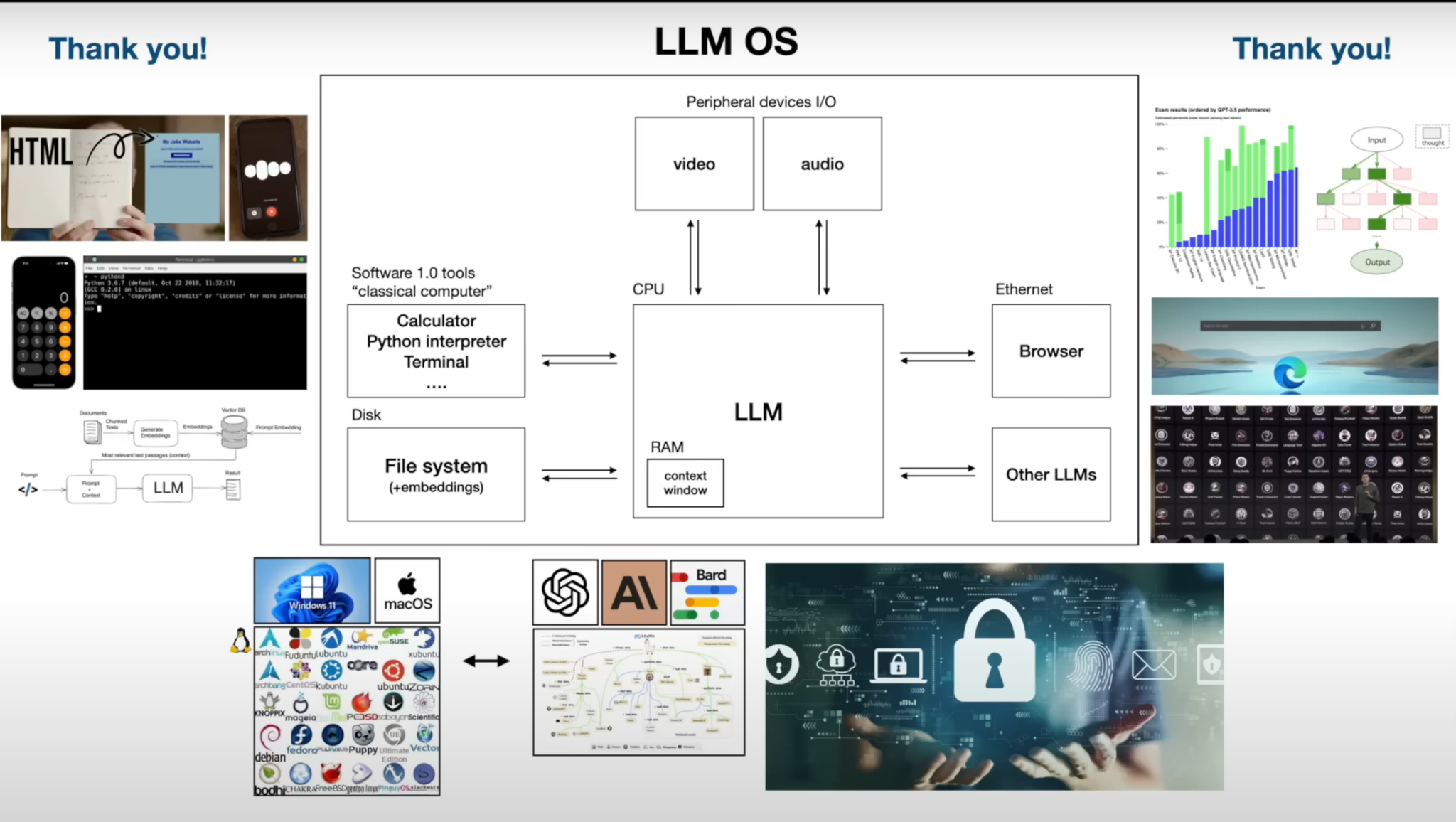
What is LLM?











**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 Huginface there is a library (Transformers: it provides a set of preprocessing classes to help prepare your data for the model.)

* 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

Example of a Fast tokenizer: https://github.com/monirmo97/LLM/blob/main/Tokenizer.ipynb

**Different types of tokenization?**

**Link of GitHub**: <https://github.com/monirmo97/LLM/blob/main/Different_Type_of_tokenization.ipynb>

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.
* 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):**

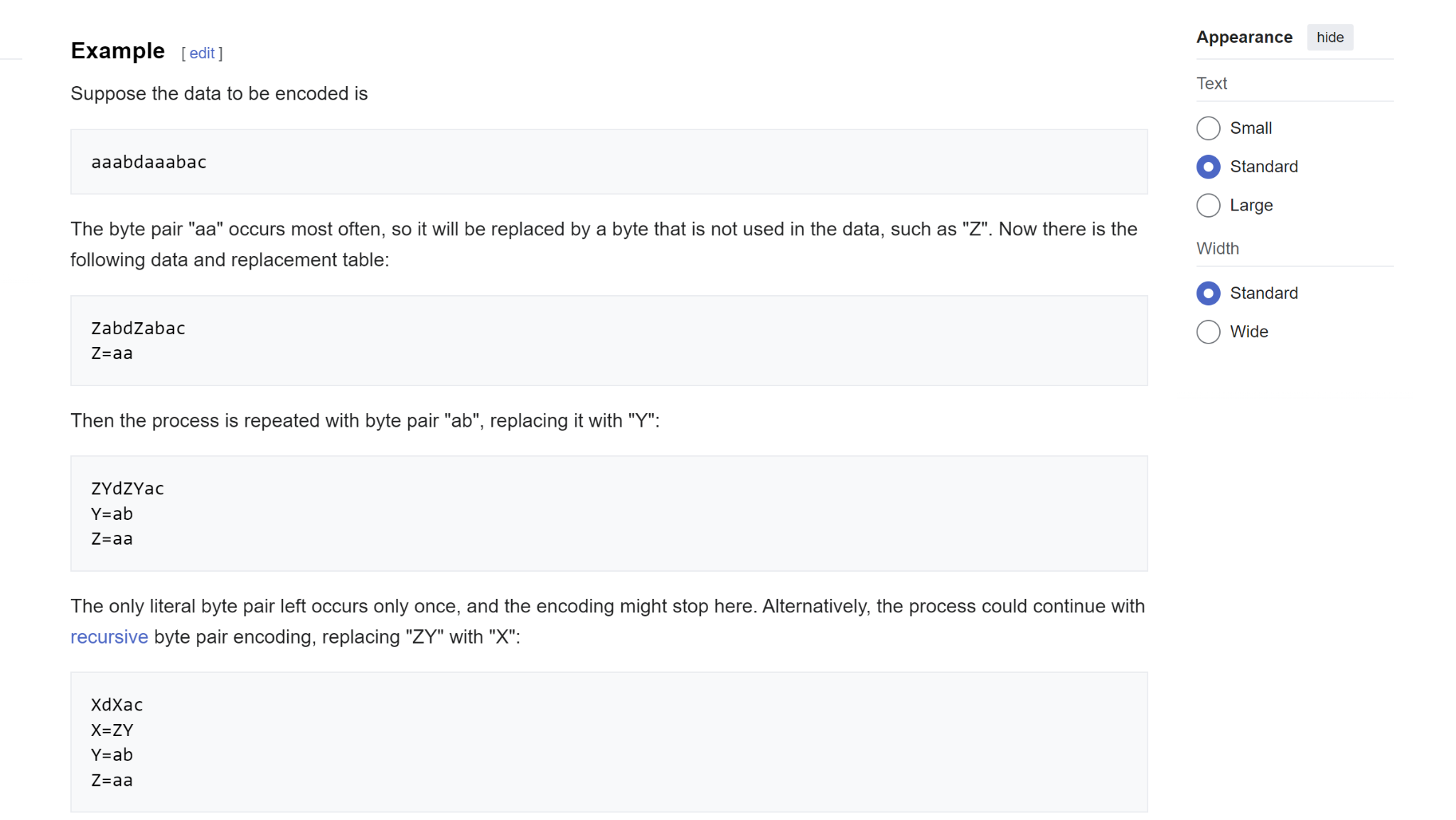
The Byte Pair Encoding (BPE) algorithm is quite instructive for understanding the basic idea of tokenization. Let's walk through an example to see how it works.

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):**

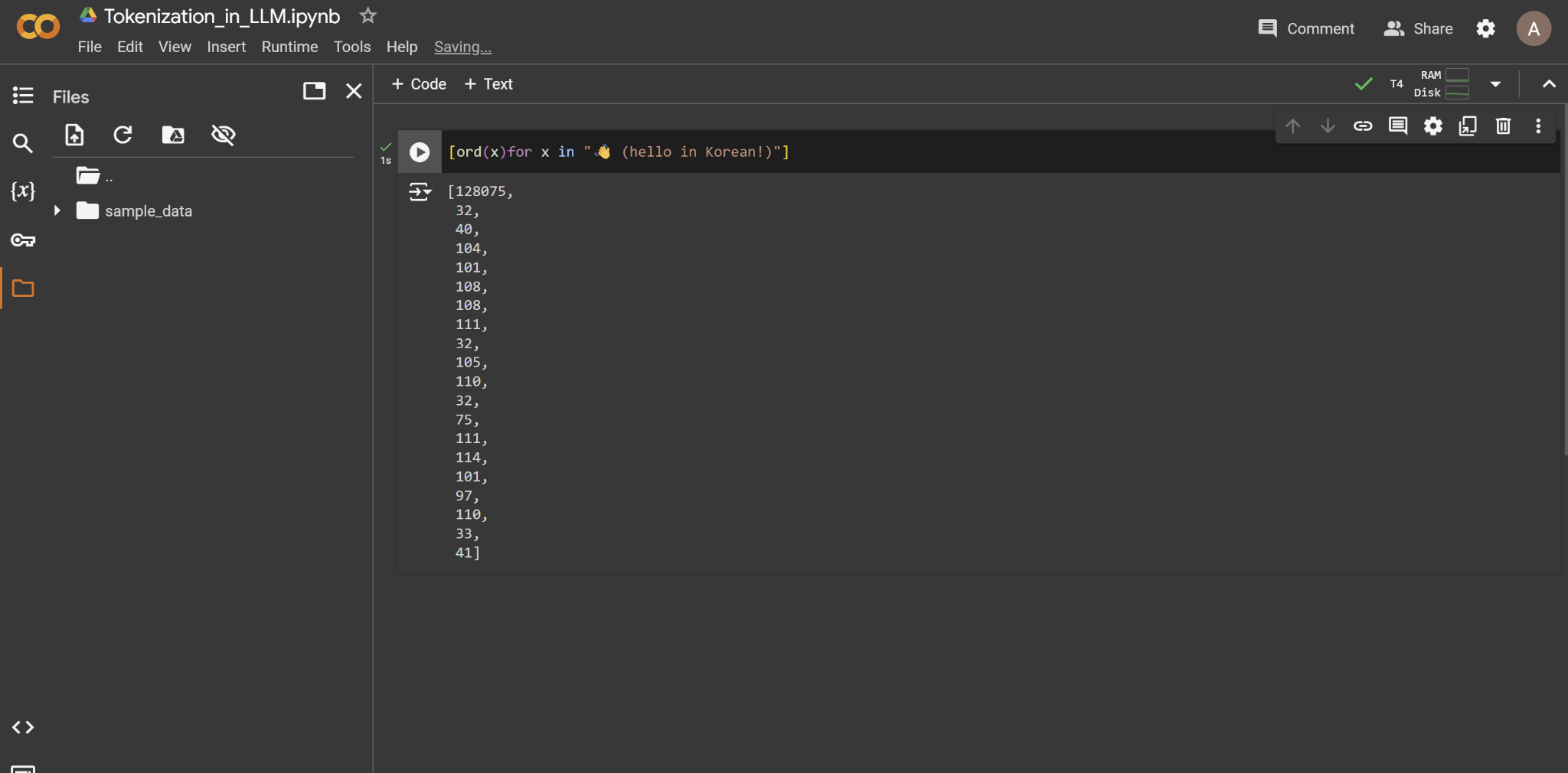
Now, I will start the implementation of tokenization in large language models (LLMs). Tokenization is a crucial component of state-of-the-art LLMs. Still, it is necessary to understand in some detail because a lot of the shining results of LLMs may be attributed to the neural network or otherwise actually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

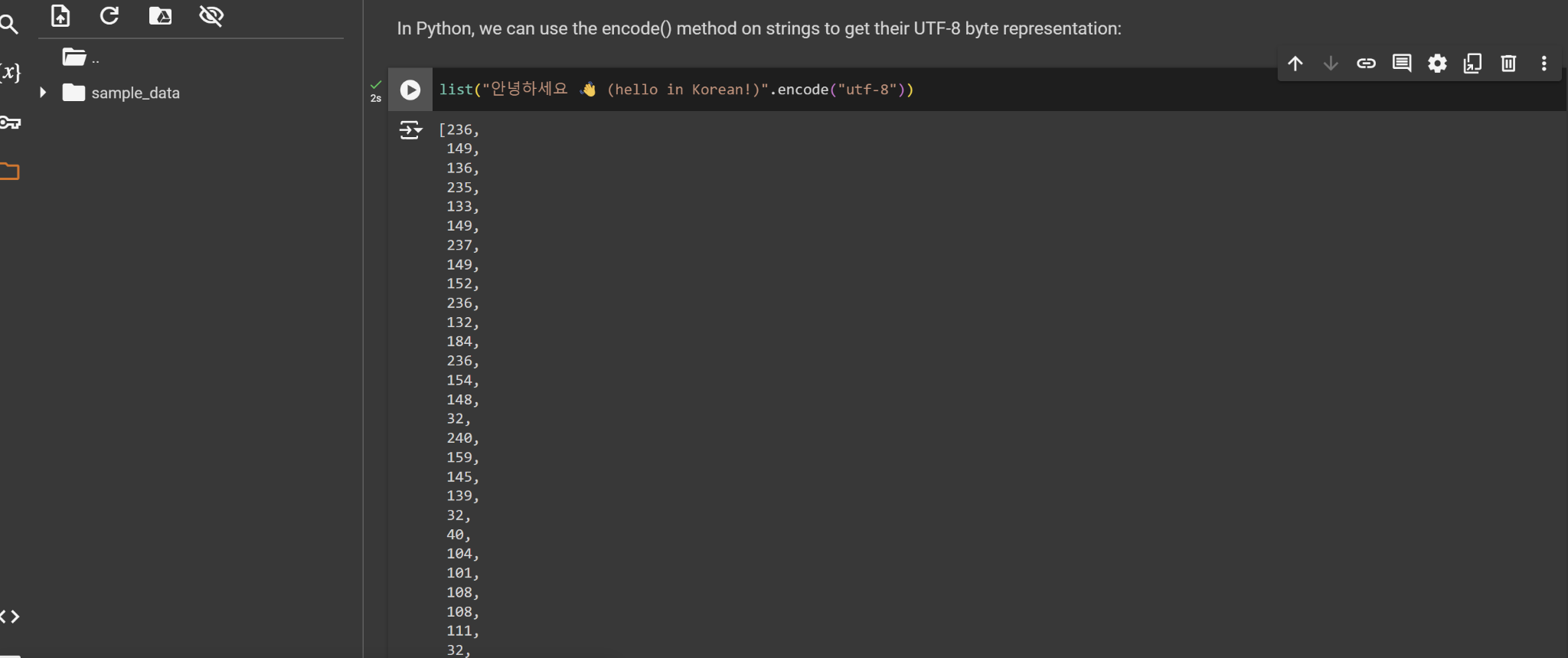
* The format is: 1110xxxx 10xxxxxx 10xxxxxx
* Example: 'ह' (U+0939)
  + Binary: 1001 0011 1001 (U+0939)
  + UTF-8: 11100000 10100100 10111001 (0xE0 0xA4 0xB9)

#### **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)

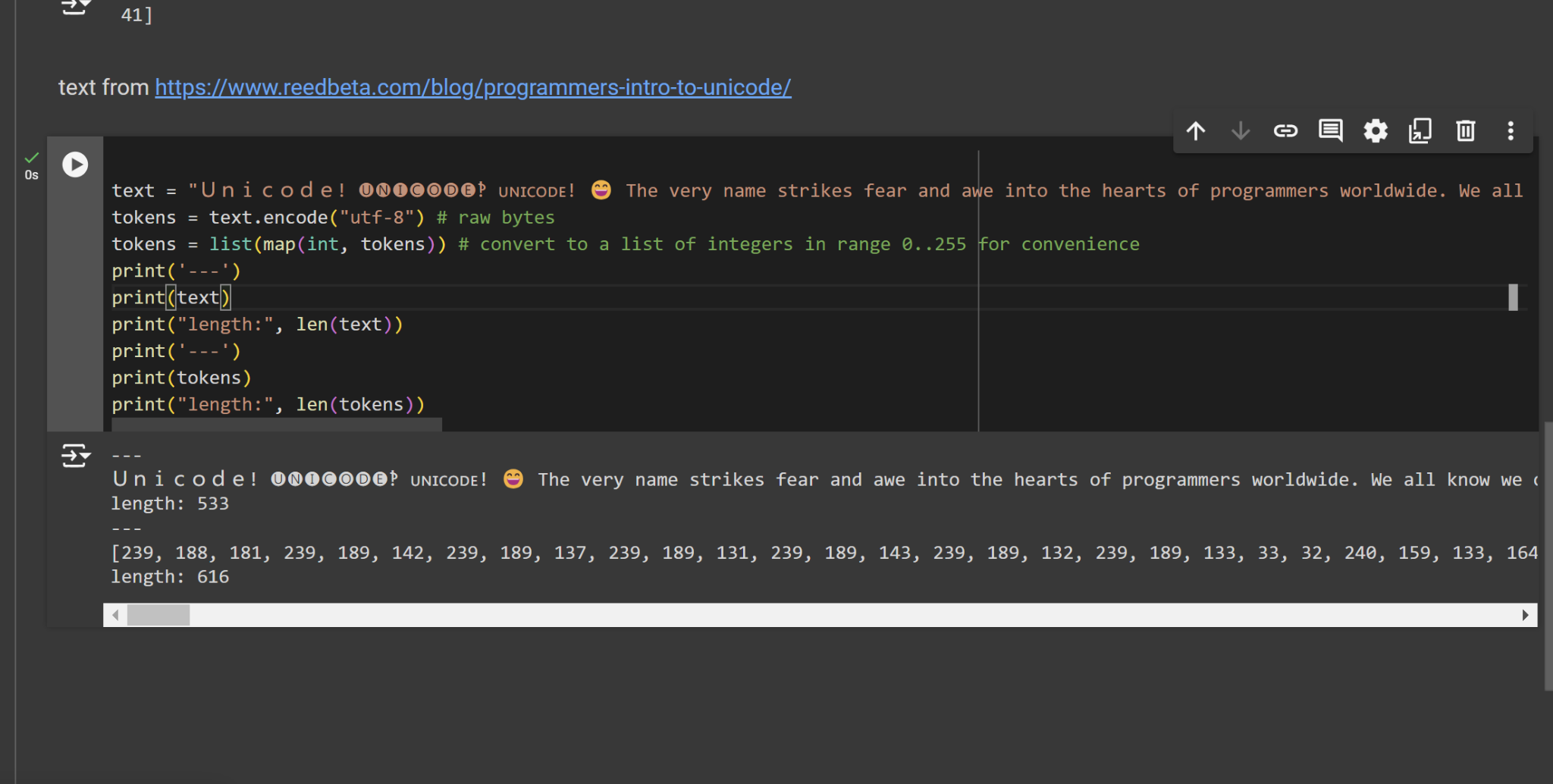
In Python, we can use the encode() method on strings to get their UTF-8 byte representation:



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.



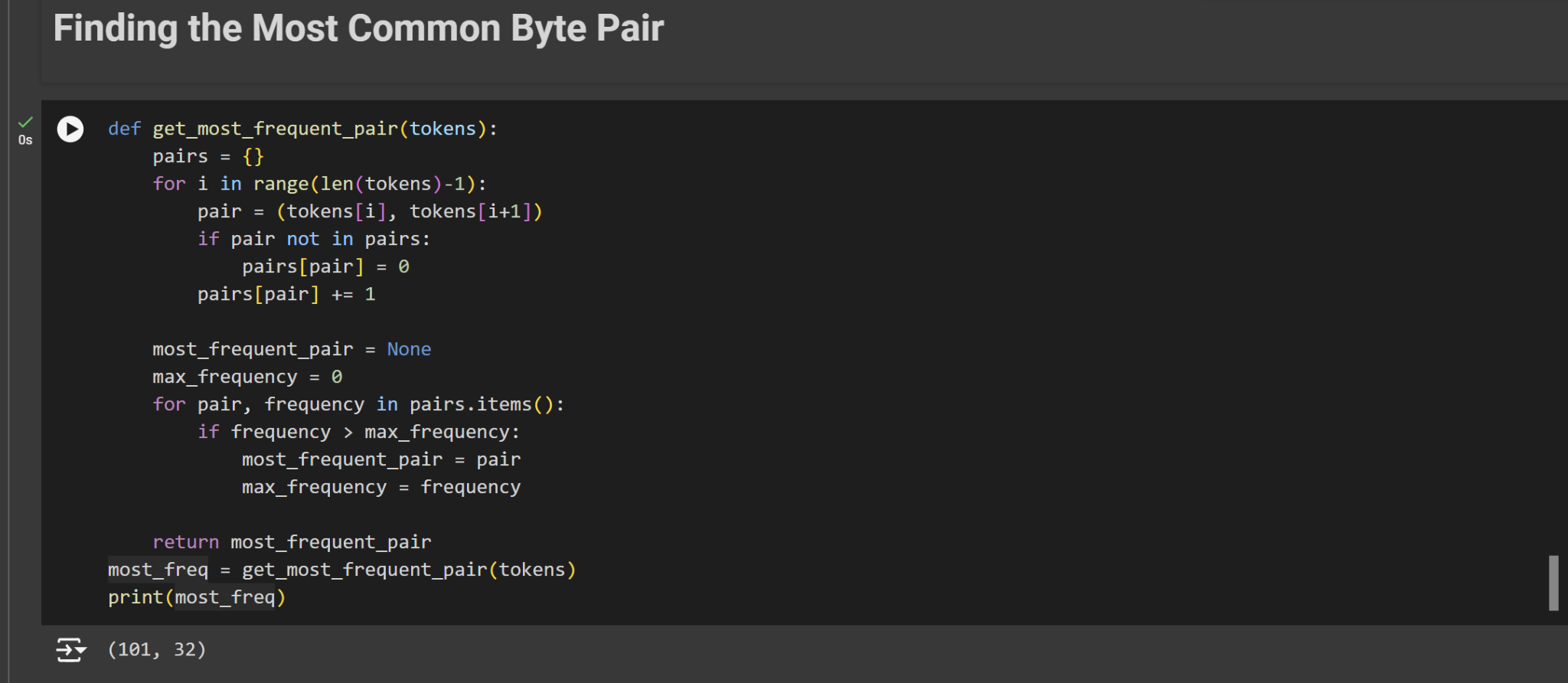
*Converting text to a list of token integers*

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/>

**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**

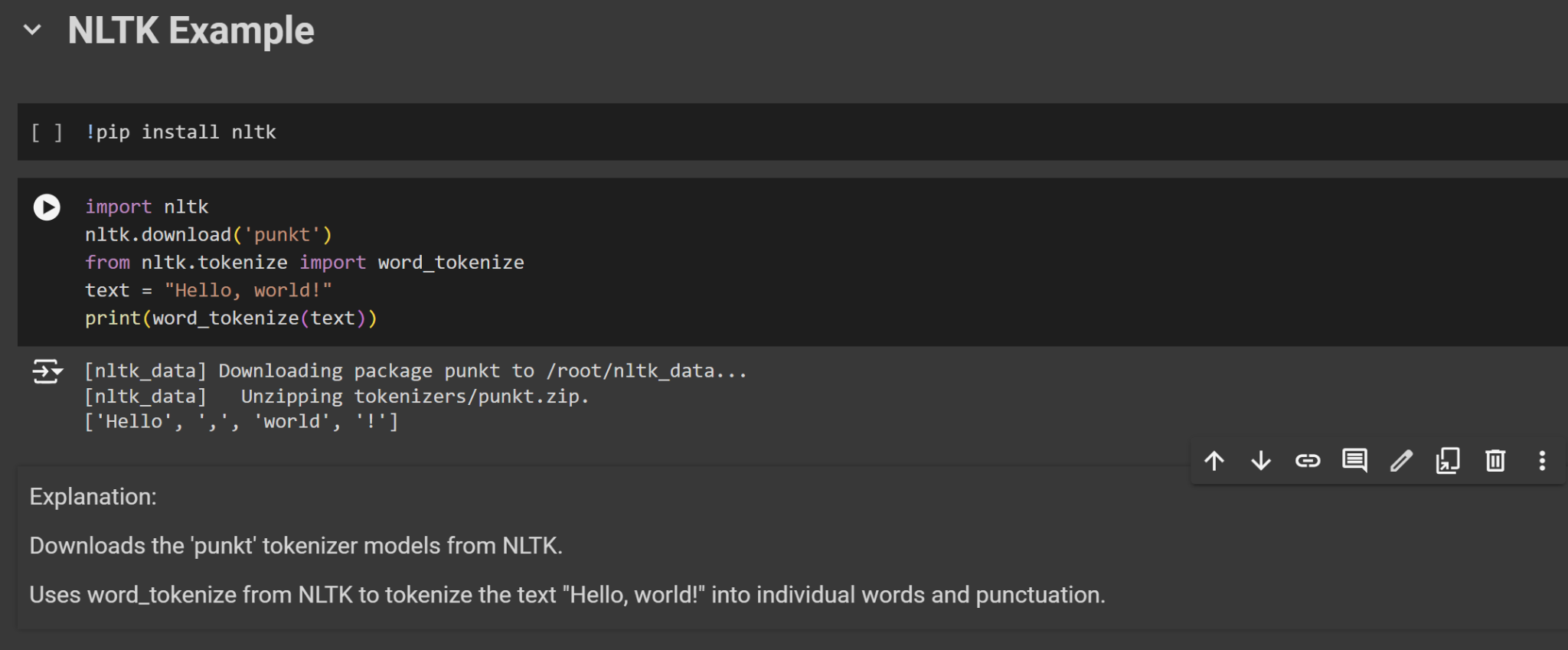
### **5. Tokenizers (by Hugging Face)**

**!pip install tokenizers**

### **6. Gensim**

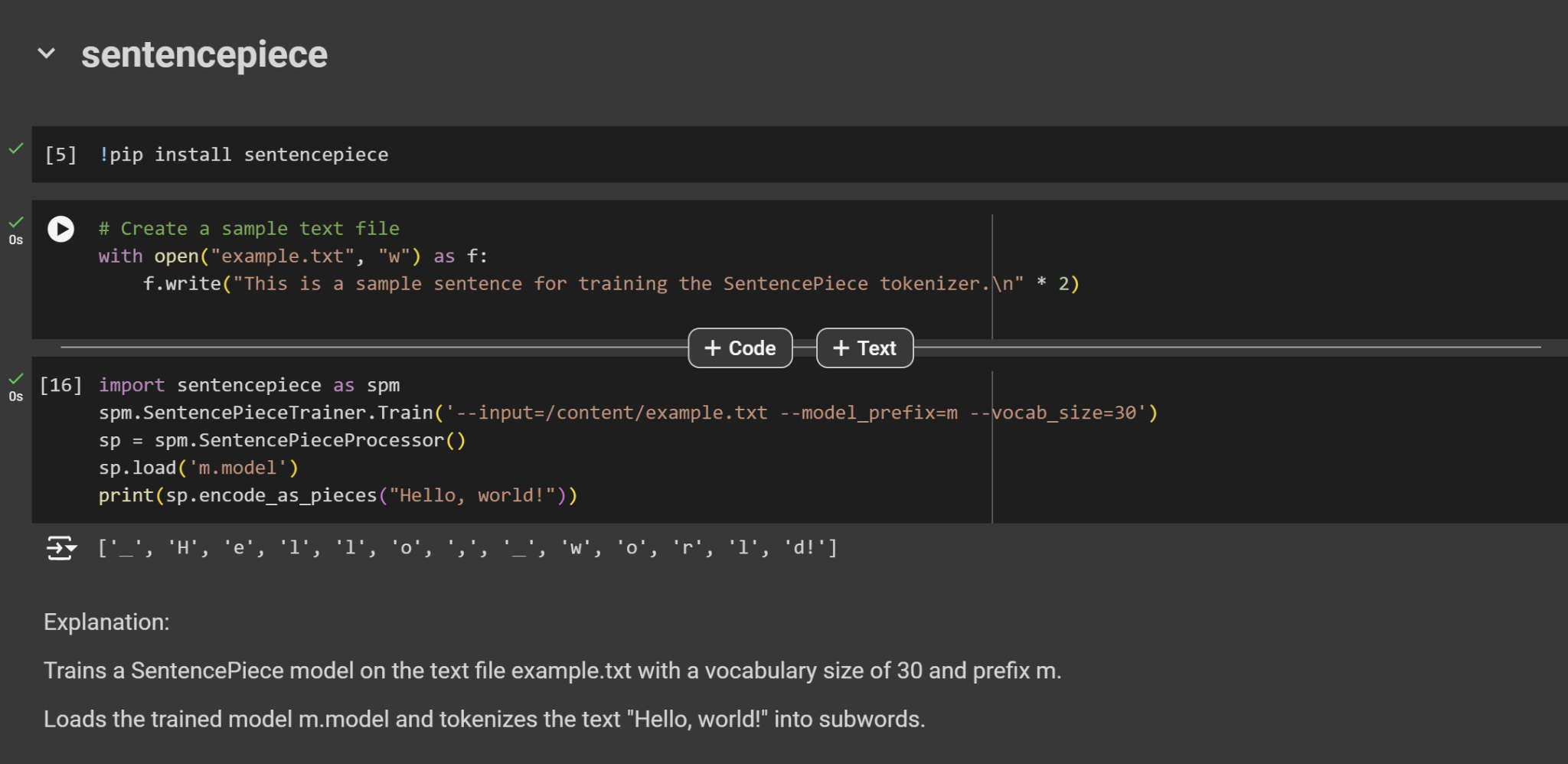
**!pip install gensim**

**Example of tokenization:**

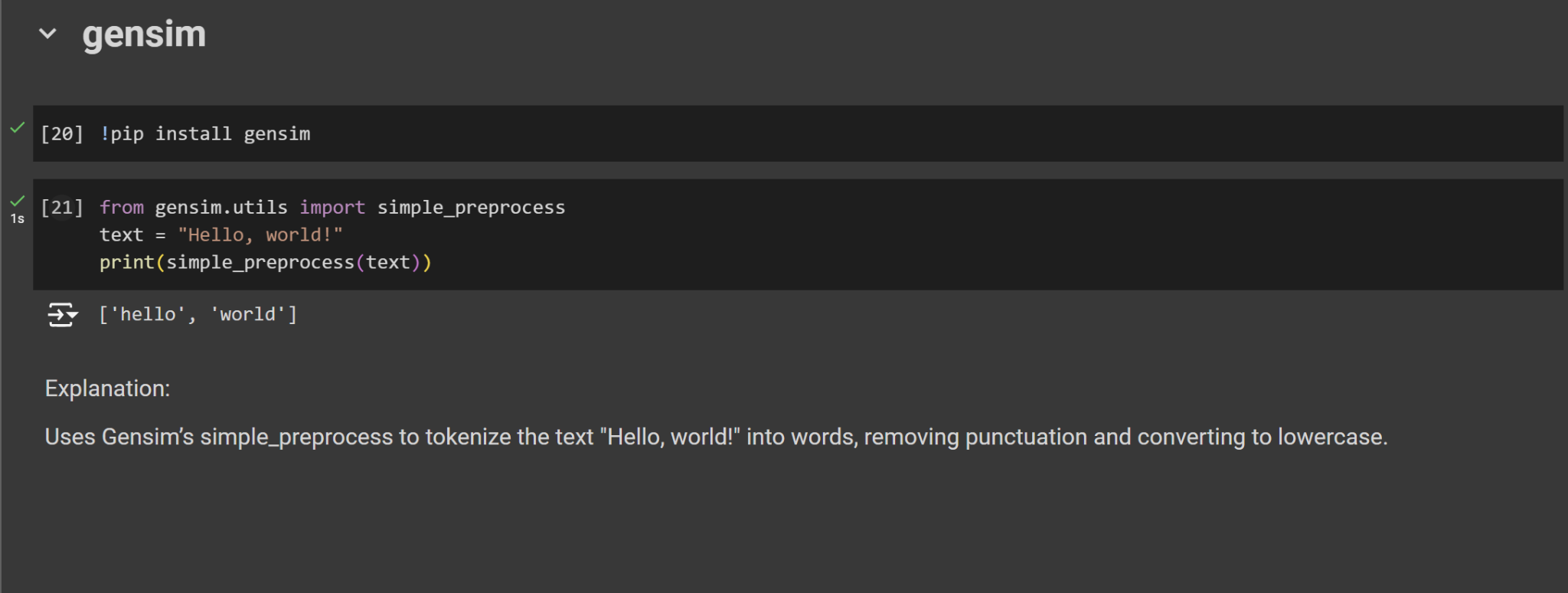
****

****

****

****

****

****

**How Can give input to LLM?**

**…..**

**Example of giving input**

**…….**

**What is prompt engineering?**

**……..**

**Example of Prompt**

**……..**

**Fine-tuning LLM**

**……….**