EMBEDDED LEARNING

SESSION 5

AUTO-COMPLETION MODEL USING ATTENTION

30/01/2025

By:

Priyanka Saxena,

Assistant Professor, CSE, KMIT.

OBJECTIVE

- The aim is to develop a model that can accurately predict and complete sequences of text or data.
- An auto-completion model using scaled dot-product attention is designed to help predict and complete sentences or sequences of text.
- Think of it like a super smart predictive text feature on your phone!!
- This is achieved by using the **scaled dot-product mechanism using Self Attention.**

Scaled Dot Product Attention

- Scaled Dot-Product Attention is a mechanism that computes attention scores between a set of queries (**Q**) and keys (**K**) using dot-product similarity, scales them, applies a softmax function, and then weights the values (**V**) accordingly.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

where:

- Q = Queries
- K = Keys
- V = Values
- d_k = Dimensionality of the keys (used for scaling)

Self-Attention

- Self-Attention is a general concept where each word (or token) in a sequence attends to every other word in the same sequence.
- It computes attention weights using **Scaled Dot-Product Attention** but specifically in a setting where **Q, K, and V come from the same input sequence**.

Summary

- Scaled Dot-Product Attention is a mathematical operation used within attention mechanisms.
- Self-Attention is a specific use case where attention is applied within the same input sequence using Scaled Dot-Product Attention.

In simpler terms:

- **Scaled Dot-Product Attention** is the **mechanism** (the mathematical operation used to compute attention scores).
- **Self-Attention** is how that mechanism is **applied** within a model to help it focus on different parts of the same input sequence.

The cat sat on the mat

Steps:

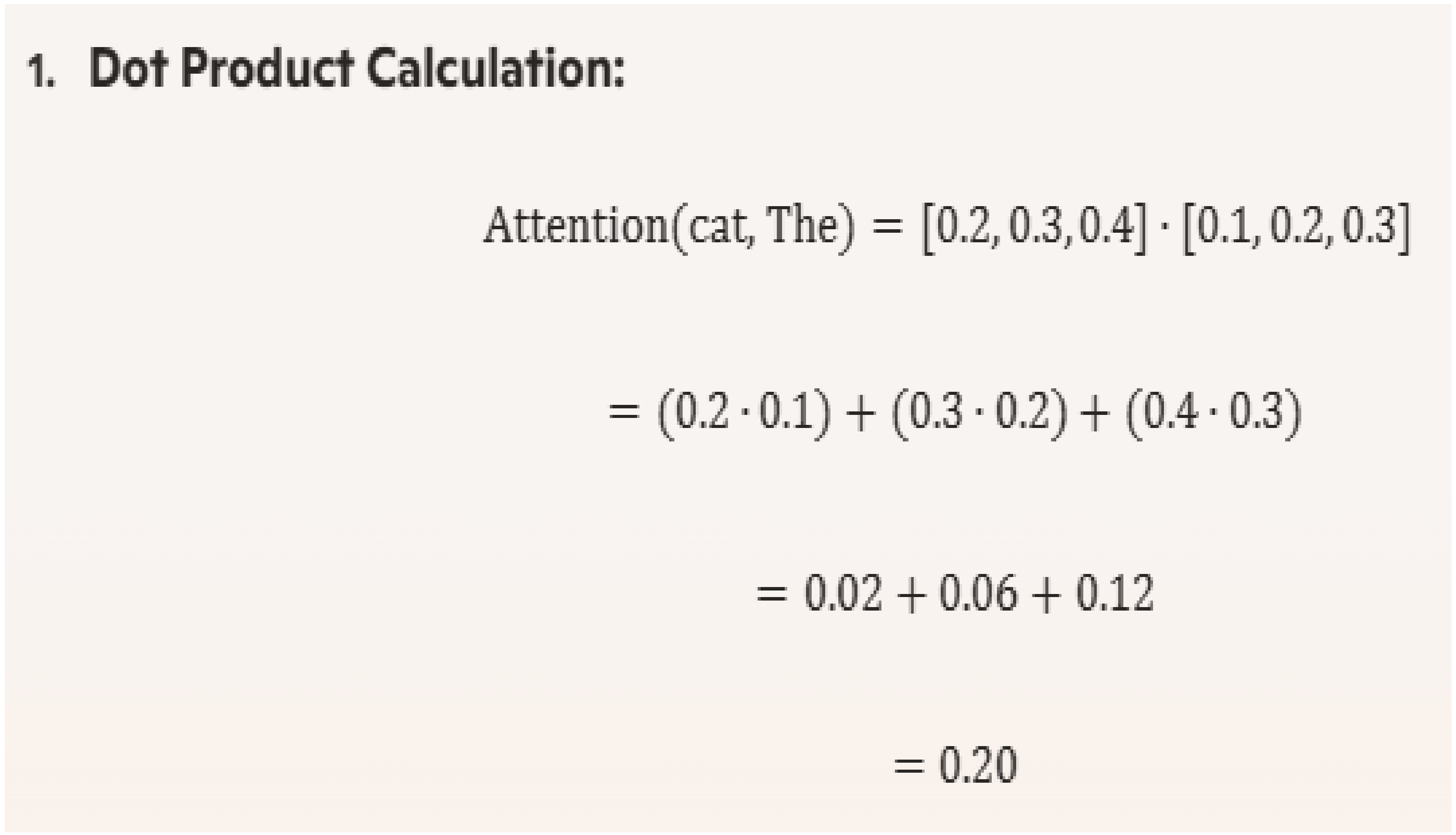
1. **Tokenization:**
    1. We split the sentence into tokens (words in this case): [The, cat, sat, on, the, mat]
2. **Assigning Vectors:**
    1. Each token is represented as a vector in an embedding space. For simplicity, let's assume each word is represented by a 3-dimensional vector.
3. **Self-Attention Calculation:**
    1. For each token, we calculate the dot product attention with all other tokens.

Word Embeddings

Let's assume we have the following embeddings for each word:

- The: [0.1, 0.2, 0.3]
- cat: [0.2, 0.3, 0.4]
- sat: [0.4, 0.5, 0.6]
- on: [0.1, 0.3, 0.2]
- the: [0.1, 0.2, 0.3]
- mat: [0.3, 0.2, 0.1]

Dot Product Attention Calculation



1. **Dot Product Calculation:**

$$\text{Attention}(\text{cat}, \text{The}) = [0.2, 0.3, 0.4] \cdot [0.1, 0.2, 0.3]$$

$$= (0.2 \cdot 0.1) + (0.3 \cdot 0.2) + (0.4 \cdot 0.3)$$

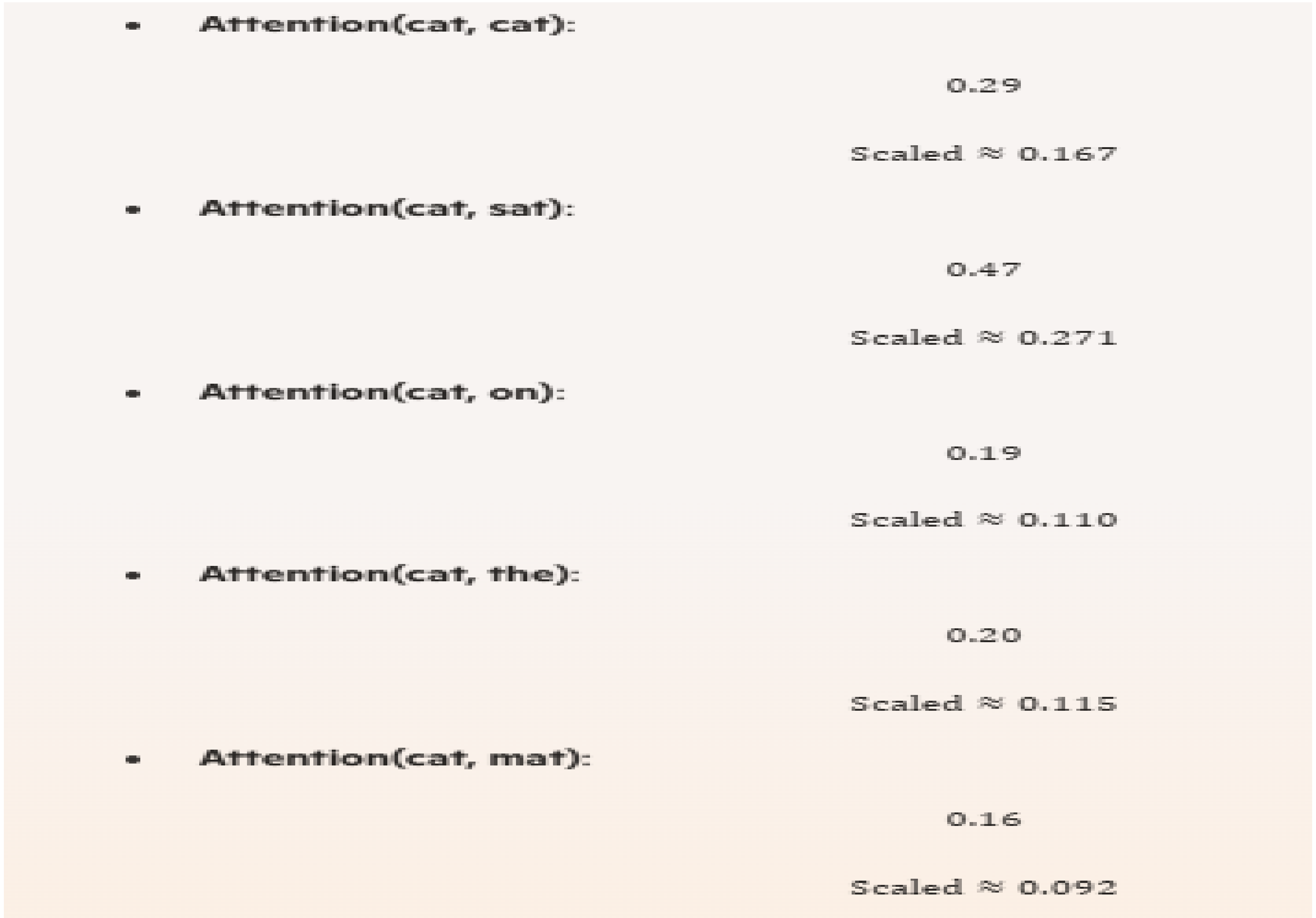
$$= 0.02 + 0.06 + 0.12$$

$$= 0.20$$

Scaling & Attention

$$\text{Scaled Attention}(\text{cat}, \text{The}) = \frac{0.20}{\sqrt{d_k}} = \frac{0.20}{\sqrt{3}} \approx 0.115$$

Remaining words / tokens



- Attention(cat, cat):

  0.29

  Scaled ≈ 0.167

- Attention(cat, sat):

  0.47

  Scaled ≈ 0.271

- Attention(cat, on):

  0.19

  Scaled ≈ 0.110

- Attention(cat, the):

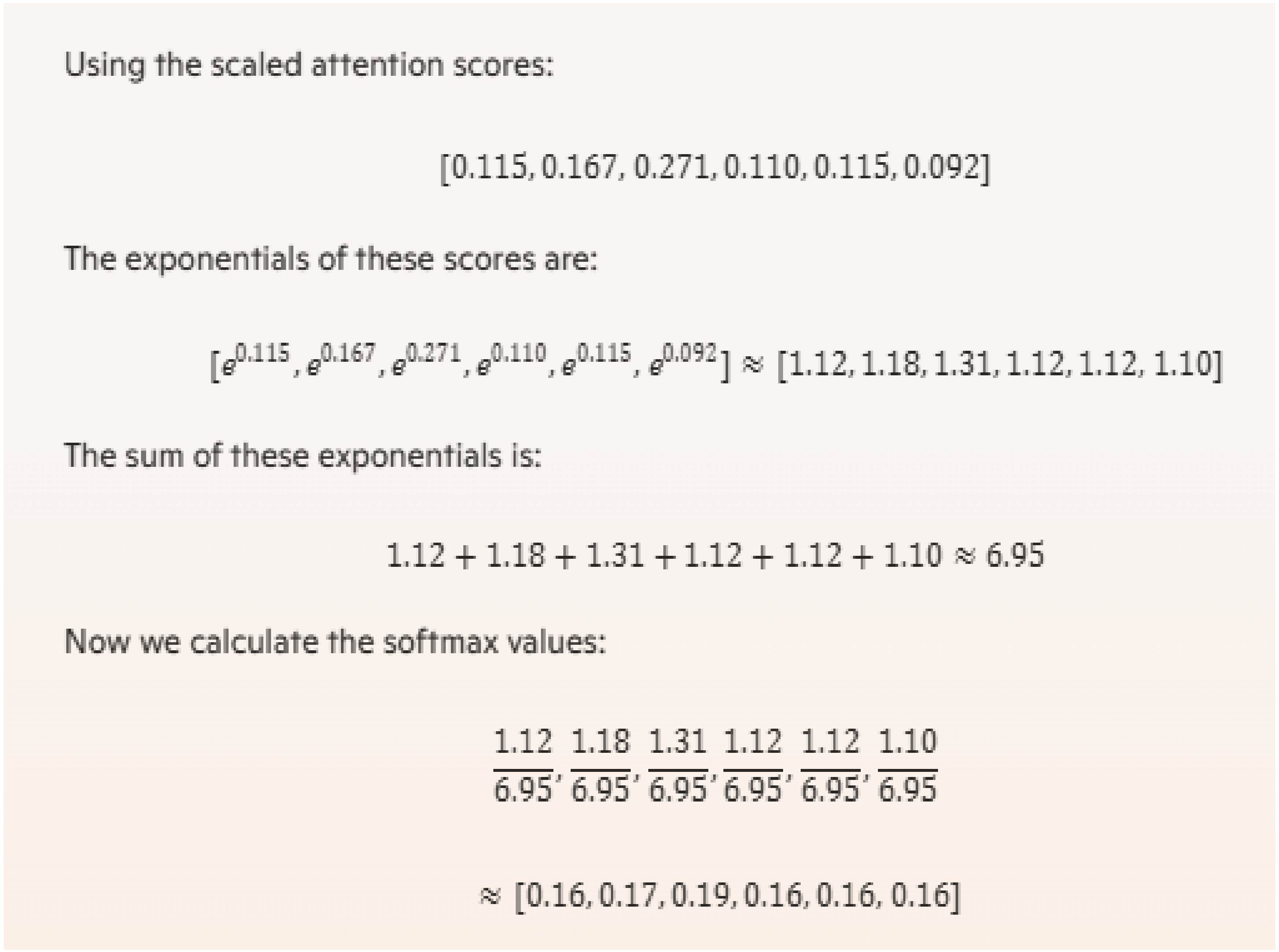
  0.20

  Scaled ≈ 0.115

- Attention(cat, mat):

  0.16

  Scaled ≈ 0.092



Using the scaled attention scores:

$$[0.115, 0.167, 0.271, 0.110, 0.115, 0.092]$$

The exponentials of these scores are:

$$[e^{0.115}, e^{0.167}, e^{0.271}, e^{0.110}, e^{0.115}, e^{0.092}] \approx [1.12, 1.18, 1.31, 1.12, 1.12, 1.10]$$

The sum of these exponentials is:

$$1.12 + 1.18 + 1.31 + 1.12 + 1.12 + 1.10 \approx 6.95$$

Now we calculate the softmax values:

$$\frac{1.12}{6.95}, \frac{1.18}{6.95}, \frac{1.31}{6.95}, \frac{1.12}{6.95}, \frac{1.12}{6.95}, \frac{1.10}{6.95}$$

$$\approx [0.16, 0.17, 0.19, 0.16, 0.16, 0.16]$$

Conclusion after softmax

- These are the scaled attention weights for the token "cat". The highest weight is for "sat," indicating the strongest attention score.

Code Overview

Data Preparation	• Import necessary libraries • Load dataset • Simple text cleaning • Covert text to integer sequences
Data Loader setup	• Test & train Split • Use PyTorch Data Loader for batch & shuffling
Scaled dot product attention class	• Attention scores & weights • Application of softmax function

Code Overview

Implement Auto Completion Model

- Embedding layer
- LSTM
- Attention Mechanism
- Fully connected layer

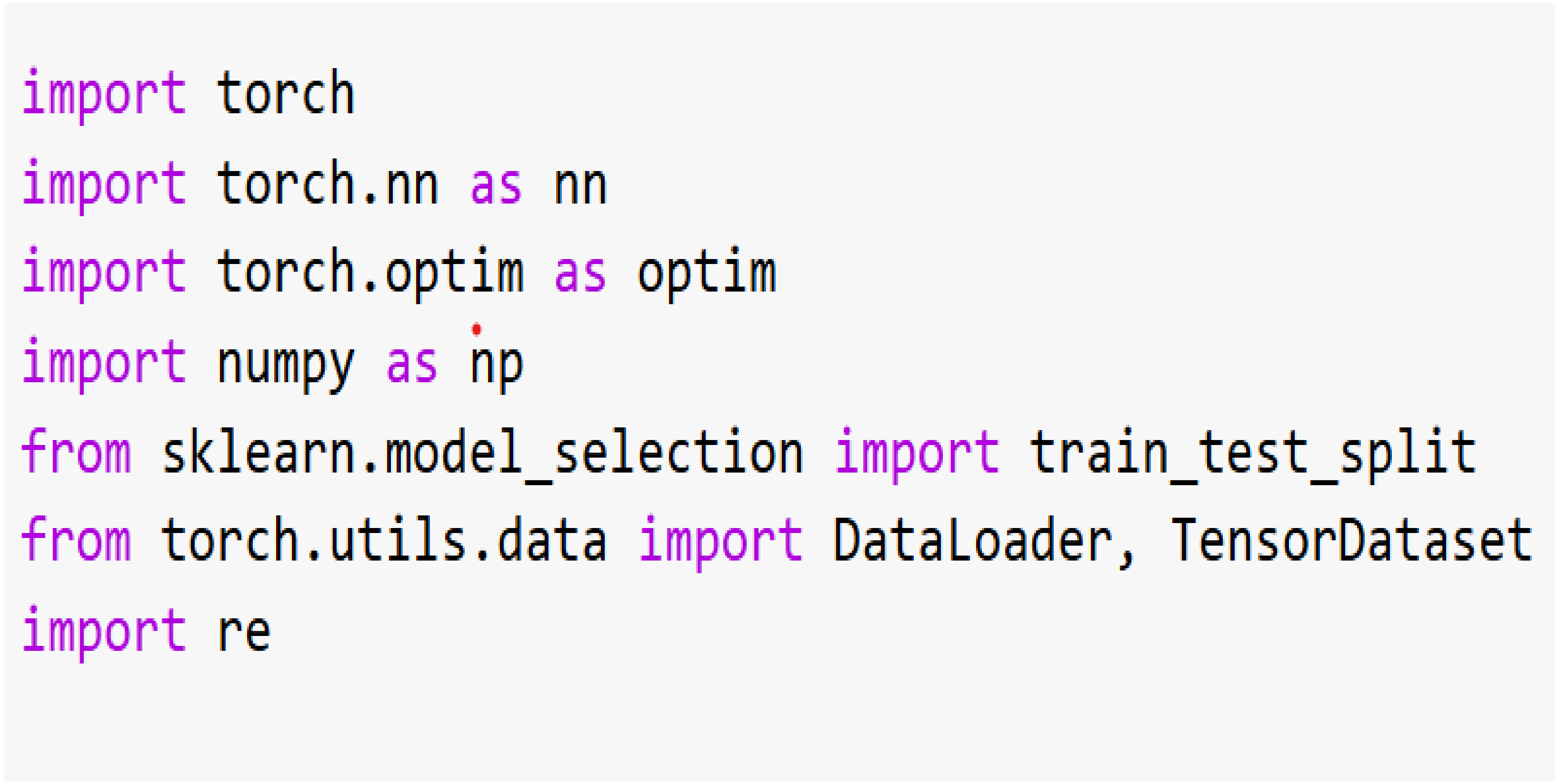
Training the model

- Loss function
- Optimiser
- Training model over suitable epochs & estimate the loss

Testing the model

- Test for autocompletion of a sample sentence

IMPORTING LIBRARIES



```python
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
from sklearn.model_selection import train_test_split
from torch.utils.data import DataLoader, TensorDataset
import re
```

Explanation of each import

Import	Explanation
torch	Core PyTorch library for tensor operations and deep learning.
torch.nn as nn	Provides classes for building neural networks (e.g., layers, activations, loss functions).
torch.optim as optim	Includes optimizers like SGD and Adam for training models.
numpy as np	Used for numerical operations and handling arrays efficiently.
sklearn.model_selection.train_test_split	Splits data into training and testing sets for model evaluation.
torch.utils.data.DataLoader	Helps efficiently load and batch data during training.
torch.utils.data.TensorDataset	Wraps tensors into a dataset, making them compatible with DataLoader.
re	Regular expressions module (likely used for text processing).

How Does `DataLoader` Work?

The `DataLoader` wraps a dataset (e.g., `TensorDataset` or `Dataset`) and provides:

✅ **Batching** – Splits data into mini-batches for efficient training.

✅ **Shuffling** – Randomly rearranges data to prevent learning order bias.

✅ **Parallel Processing** – Uses multiple workers (`num_workers`) for faster data loading.

✅ **Automatic Iteration** – Provides an easy way to loop through dataset batches.

Why Use `TensorDataset`?

✅ It helps **organize data** when working with PyTorch models.

✅ Makes it easier to **batch, shuffle, and sample** data.

✅ Works well with `DataLoader` to **streamline training**.

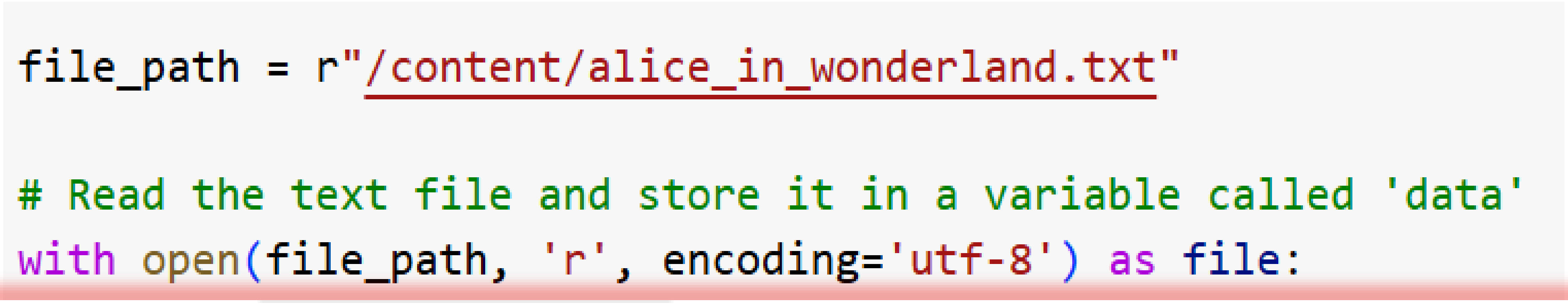
Why Use `re`?

✅ **Pattern Matching** – Find specific words, numbers, or formats in text.

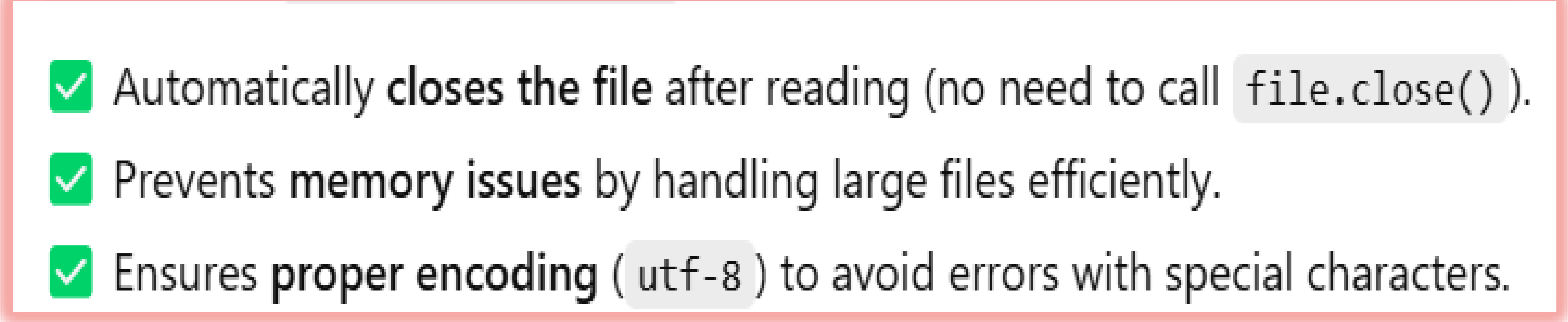
✅ **Text Cleaning** – Remove unwanted characters (e.g., punctuation, whitespace).

✅ **Validation** – Check if text follows a specific format (e.g., email, phone number).

Load the dataset



```python
file_path = r"/content/alice_in_wonderland.txt"

# Read the text file and store it in a variable called 'data'
with open(file_path, 'r', encoding='utf-8') as file:
```



- ✅ Automatically **closes the file** after reading (no need to call `file.close()`).
- ✅ Prevents **memory issues** by handling large files efficiently.
- ✅ Ensures **proper encoding** (`utf-8`) to avoid errors with special characters.

```python
file_path = r"/content/alice_in_wonderland.txt"

# Read the text file and store it in a variable called 'data'
with open(file_path, 'r', encoding='utf-8') as file:
    data = file.read()

                                                          ext
```

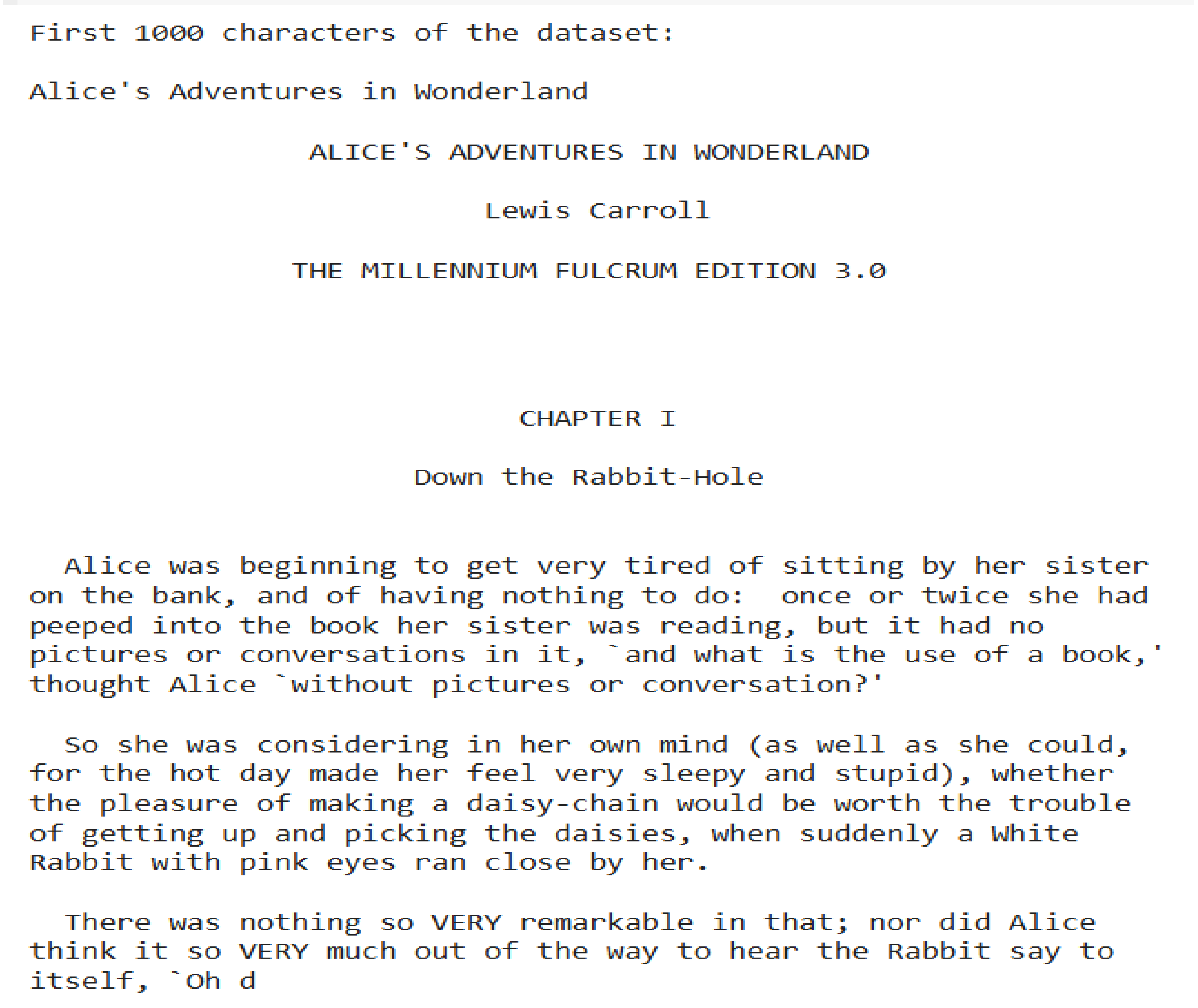
Reading the file:

- `with open(file_path, 'r', encoding='utf-8') as file`: This opens the file in read mode (`'r'`) with UTF-8 encoding, ensuring that it can handle any special characters in the file.
- `data = file.read()`: Reads the entire content of the file into the variable `data`.

```python
file_path = r"/content/alice_in_wonderland.txt"

# Read the text file and store it in a variable called 'data'
with open(file_path, 'r', encoding='utf-8') as file:
    data = file.read()

# Print a sample of the dataset
print("First 1000 characters of the dataset:\n")
print(data[:1000])  # Show the first 1000 characters of the text
```

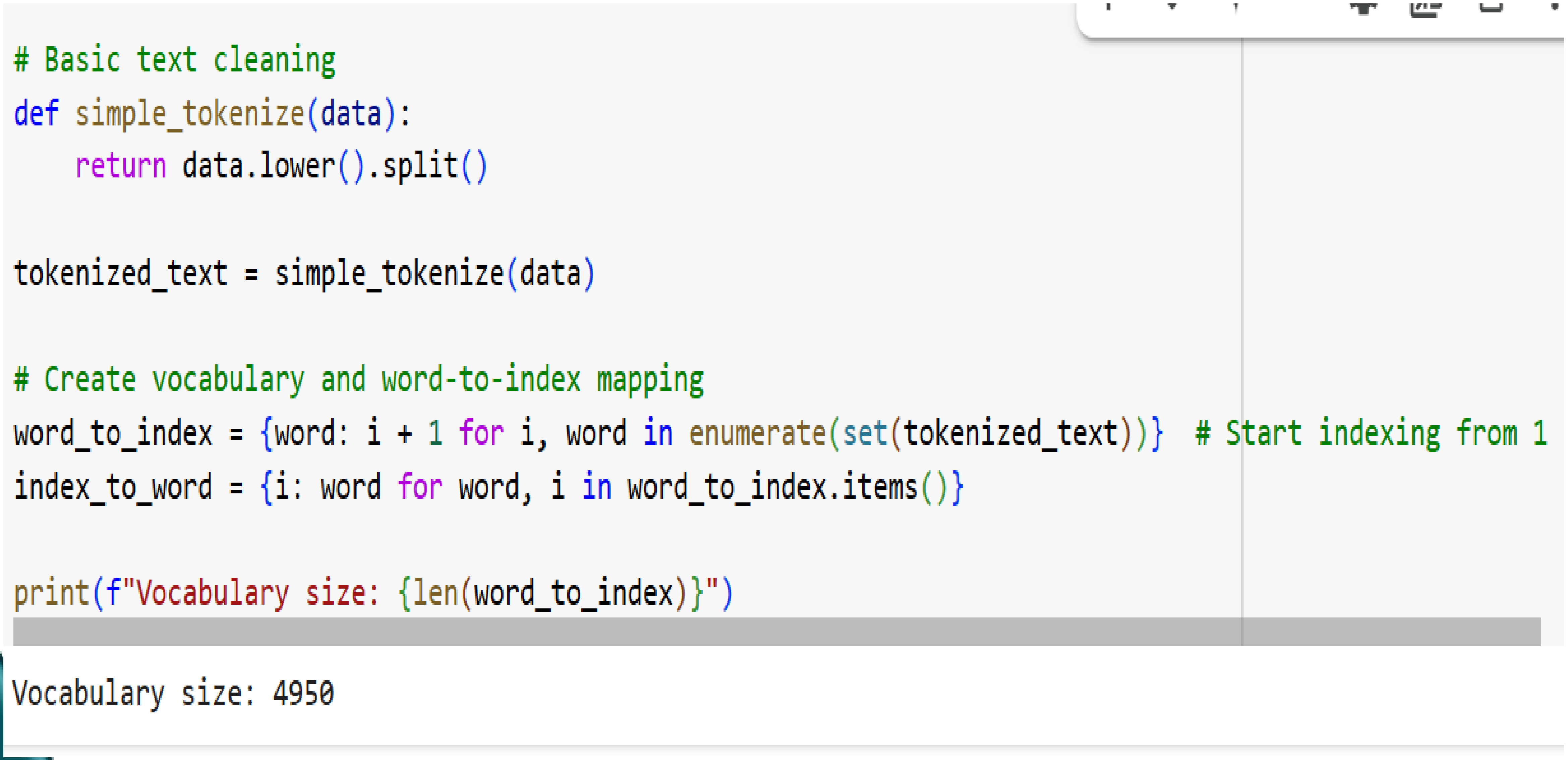
Printing a sample:

- `print(data[:1000])` : This will print the first 1000 characters from the text file, giving you a preview of the content.



```
First 1000 characters of the dataset:

Alice's Adventures in Wonderland

                ALICE'S ADVENTURES IN WONDERLAND

                          Lewis Carroll

               THE MILLENNIUM FULCRUM EDITION 3.0




                            CHAPTER I

                      Down the Rabbit-Hole


  Alice was beginning to get very tired of sitting by her sister
on the bank, and of having nothing to do:  once or twice she had
peeped into the book her sister was reading, but it had no
pictures or conversations in it, `and what is the use of a book,'
thought Alice `without pictures or conversation?'

  So she was considering in her own mind (as well as she could,
for the hot day made her feel very sleepy and stupid), whether
the pleasure of making a daisy-chain would be worth the trouble
of getting up and picking the daisies, when suddenly a White
Rabbit with pink eyes ran close by her.

  There was nothing so VERY remarkable in that; nor did Alice
think it so VERY much out of the way to hear the Rabbit say to
itself, `Oh d
```

Basic text cleaning



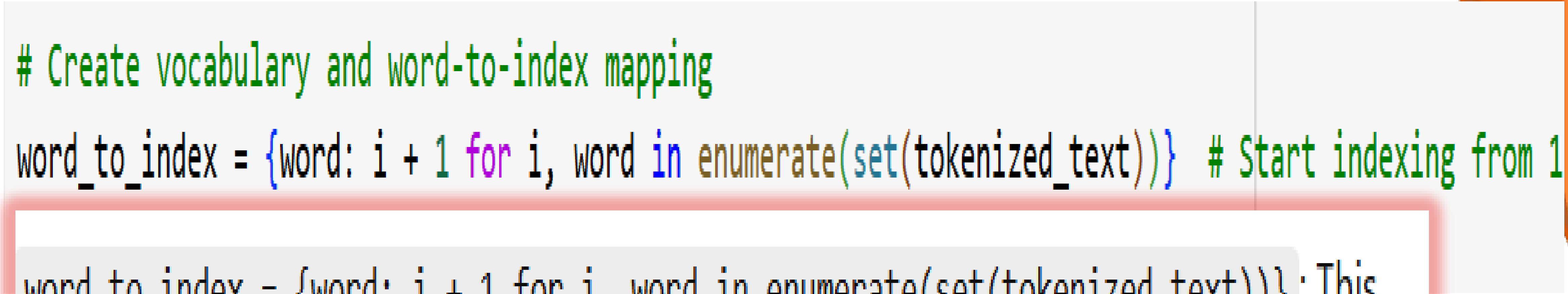
```
# Basic text cleaning
def simple_tokenize(data):
    return data.lower().split()

tokenized_text = simple_tokenize(data)

# Create vocabulary and word-to-index mapping
word_to_index = {word: i + 1 for i, word in enumerate(set(tokenized_text))}  # Start indexing from 1
index_to_word = {i: word for word, i in word_to_index.items()}

print(f"Vocabulary size: {len(word_to_index)}")
```

```
Vocabulary size: 4950
```

```python
# Basic text cleaning
def simple_tokenize(data):
    return data.lower().split()

tokenized_text = simple_tokenize(data)
```

Tokenization (`simple_tokenize`):

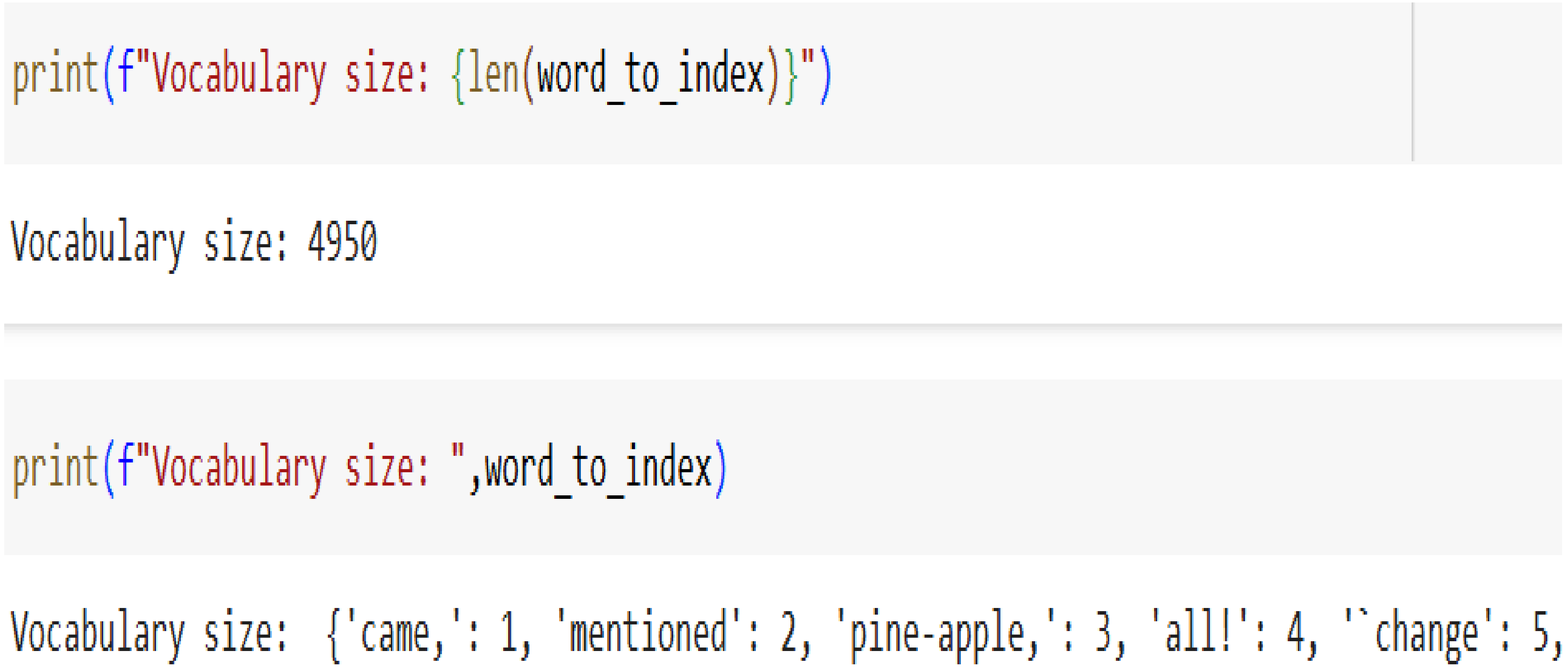
- `data.lower().split()`: Converts the text to lowercase and splits it into words based on spaces. This is a simple way to break the text into tokens (words).



```python
# Create vocabulary and word-to-index mapping
word_to_index = {word: i + 1 for i, word in enumerate(set(tokenized_text))}  # Start indexing from 1
```

`word_to_index = {word: i + 1 for i, word in enumerate(set(tokenized_text))}`: This creates a dictionary where each unique word is mapped to a unique index. The index starts from 1 (not 0) because usually, indexing starts from 1 in text processing tasks.

`index_to_word = {i: word for word, i in word_to_index.items()}`: This reverses the mapping to get an index-to-word dictionary.



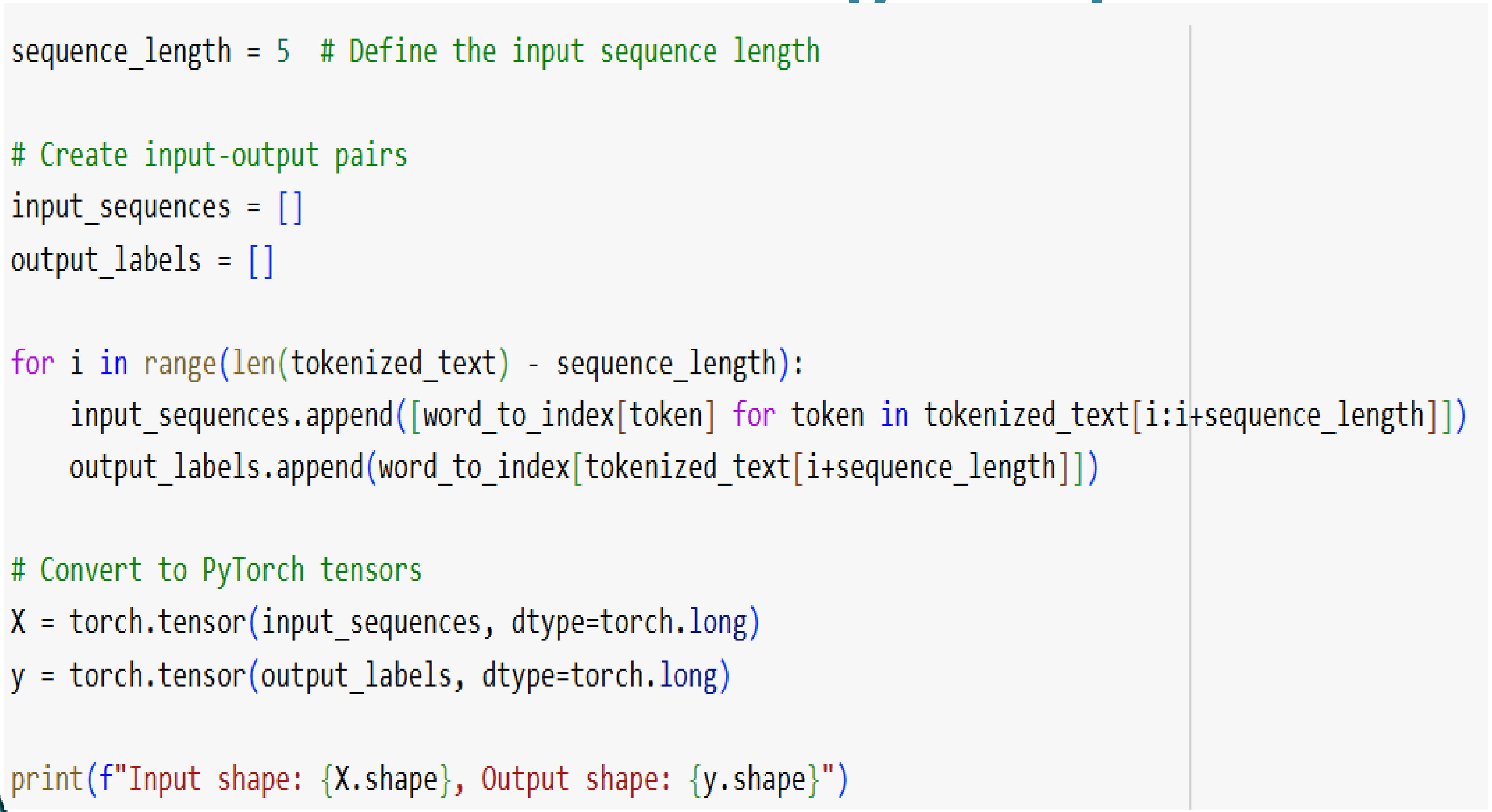
```python
print(f"Vocabulary size: {len(word_to_index)}")
```

```
Vocabulary size: 4950
```

```python
print(f"Vocabulary size: ",word_to_index)
```

```
Vocabulary size:  {'came,': 1, 'mentioned': 2, 'pine-apple,': 3, 'all!': 4, '`change': 5,
```

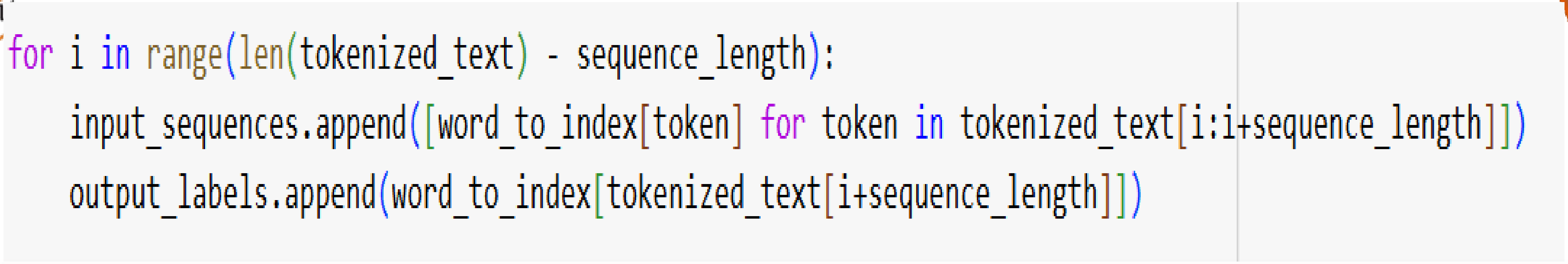
Convert Text to Integer Sequences



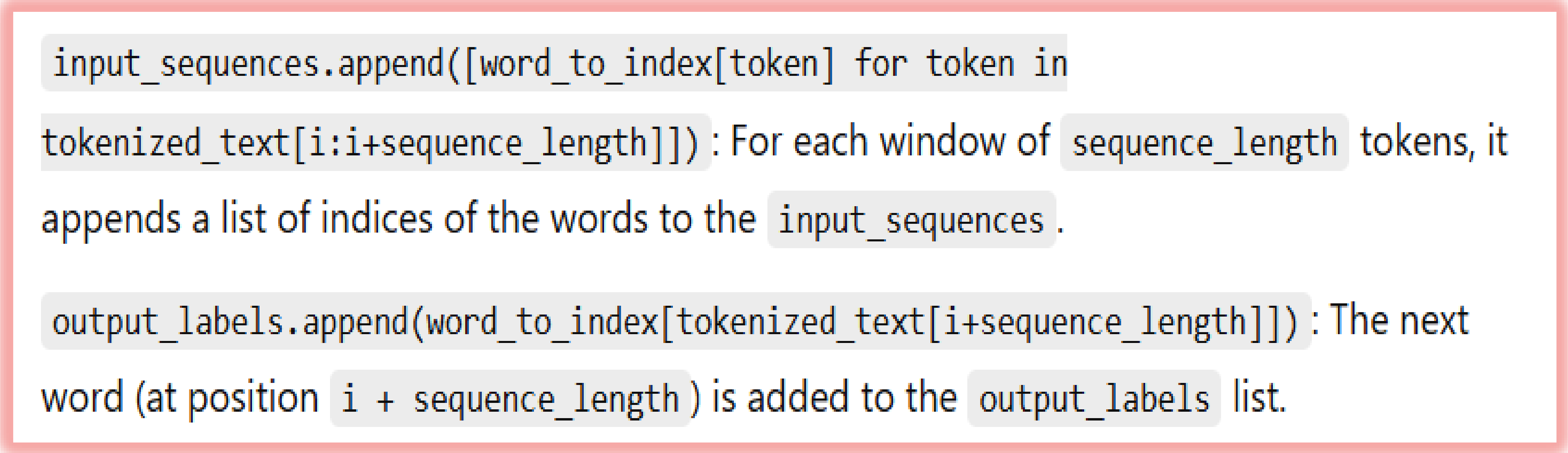
```python
sequence_length = 5  # Define the input sequence length

# Create input-output pairs
input_sequences = []
output_labels = []

for i in range(len(tokenized_text) - sequence_length):
    input_sequences.append([word_to_index[token] for token in tokenized_text[i:i+sequence_length]])
    output_labels.append(word_to_index[tokenized_text[i+sequence_length]])

# Convert to PyTorch tensors
X = torch.tensor(input_sequences, dtype=torch.long)
y = torch.tensor(output_labels, dtype=torch.long)

print(f"Input shape: {X.shape}, Output shape: {y.shape}")
```

```python
sequence_length = 5  # Define the input sequence length

# Create input-output pairs
input_sequences = []
output_labels = []

for i in range(len(tokenized_text) - sequence_length):
                                                                    ])
```

The loop `for i in range(len(tokenized_text) - sequence_length)` iterates over the tokenized text, creating sequences of length `sequence_length` for the input and the next word as the output (label).

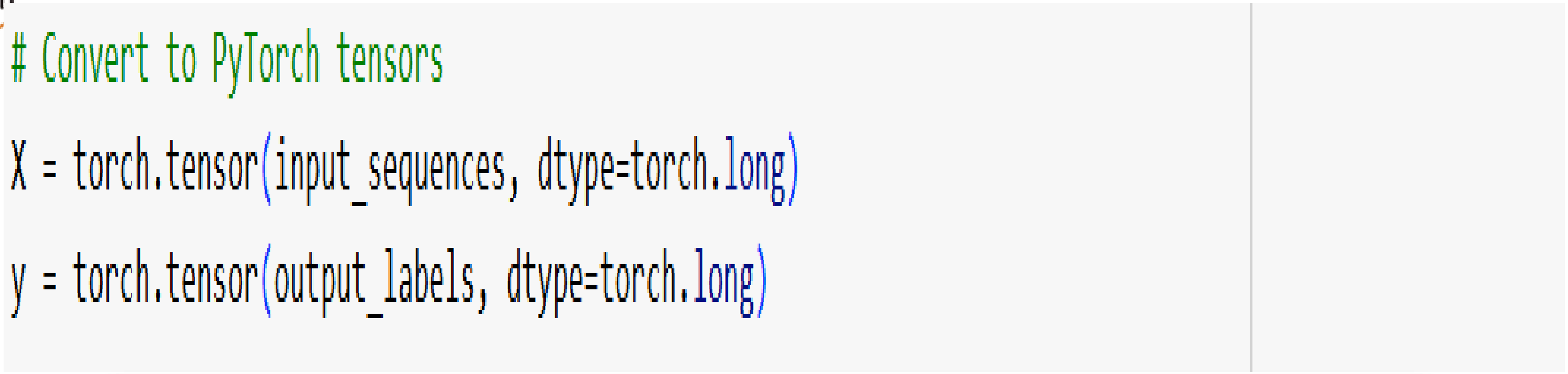


```
for i in range(len(tokenized_text) - sequence_length):
    input_sequences.append([word_to_index[token] for token in tokenized_text[i:i+sequence_length]])
    output_labels.append(word_to_index[tokenized_text[i+sequence_length]])
```



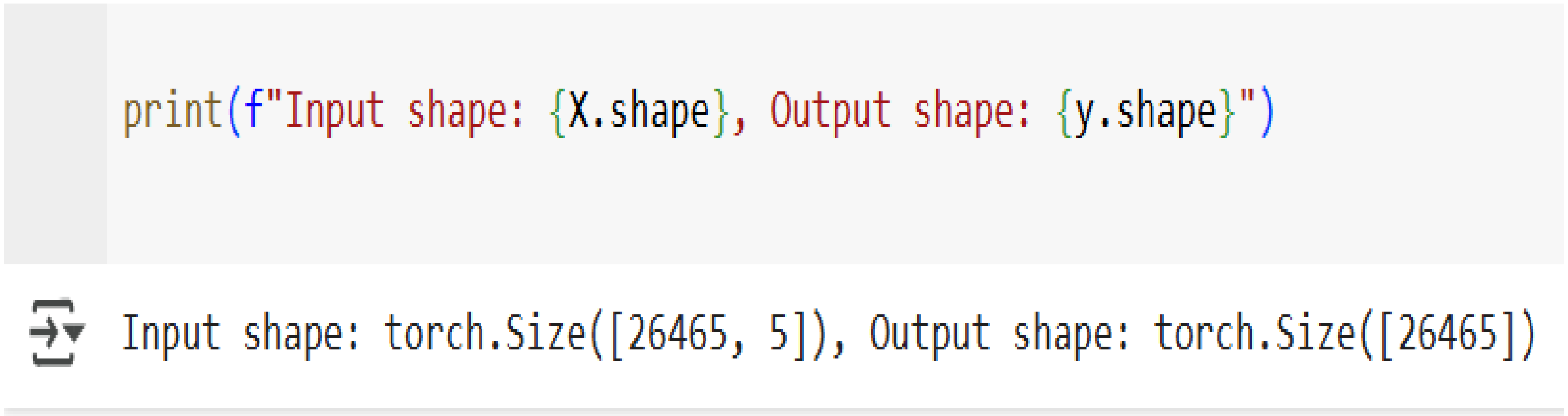
`input_sequences.append([word_to_index[token] for token in tokenized_text[i:i+sequence_length]])`: For each window of `sequence_length` tokens, it appends a list of indices of the words to the `input_sequences`.

`output_labels.append(word_to_index[tokenized_text[i+sequence_length]])`: The next word (at position `i + sequence_length`) is added to the `output_labels` list.



```
# Convert to PyTorch tensors
X = torch.tensor(input_sequences, dtype=torch.long)
y = torch.tensor(output_labels, dtype=torch.long)
```

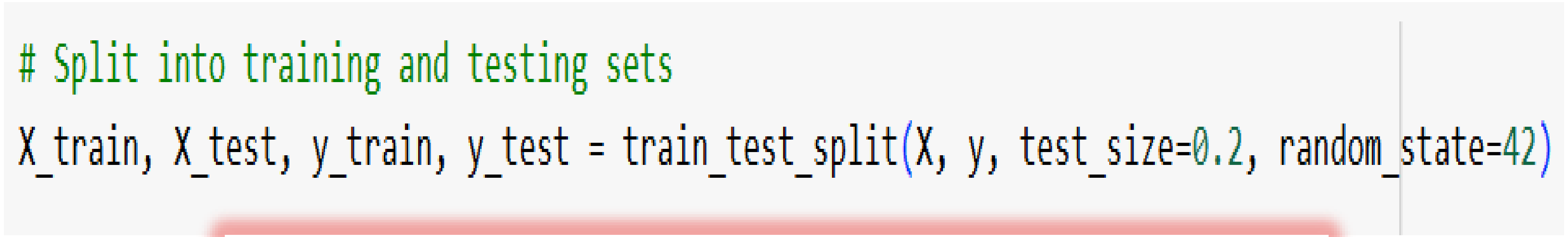
- `X = torch.tensor(input_sequences, dtype=torch.long)`: Converts the input sequences into a PyTorch tensor of type `long` (integer type).
- `y = torch.tensor(output_labels, dtype=torch.long)`: Converts the output labels into a PyTorch tensor.



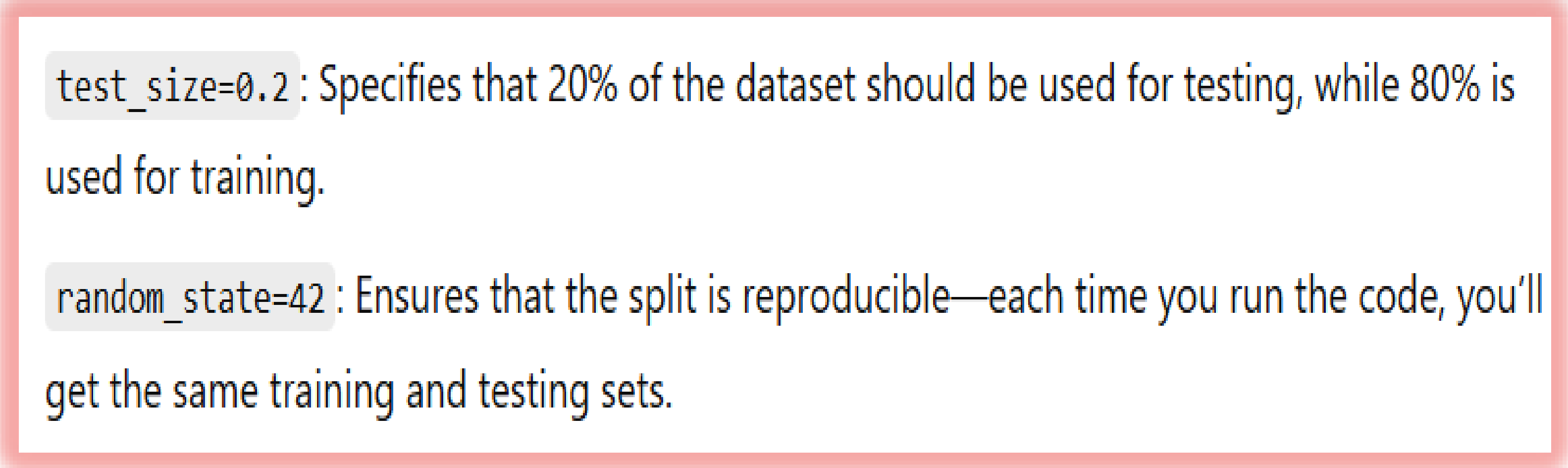
```python
print(f"Input shape: {X.shape}, Output shape: {y.shape}")
```

```
Input shape: torch.Size([26465, 5]), Output shape: torch.Size([26465])
```

Train-Test Split and DataLoader



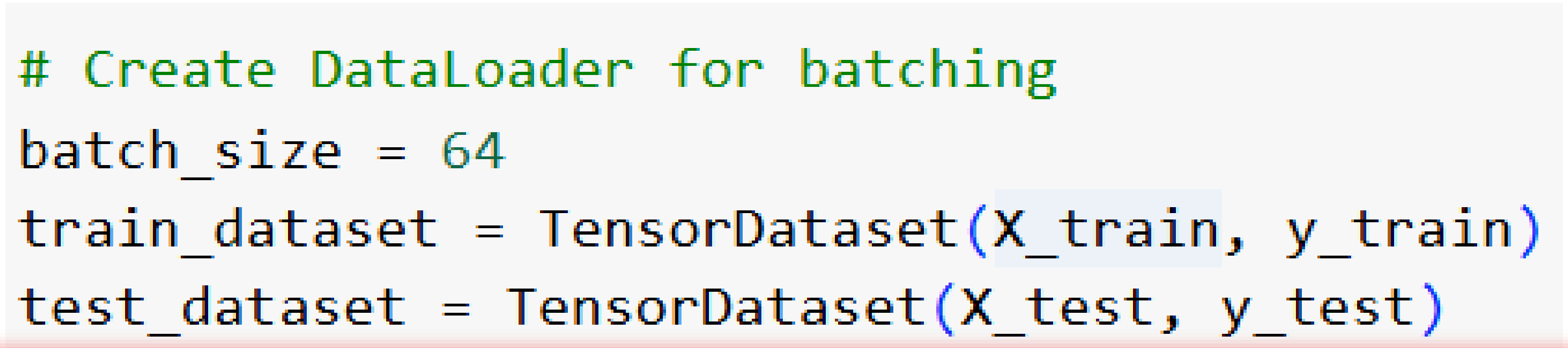
```
# Split into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```



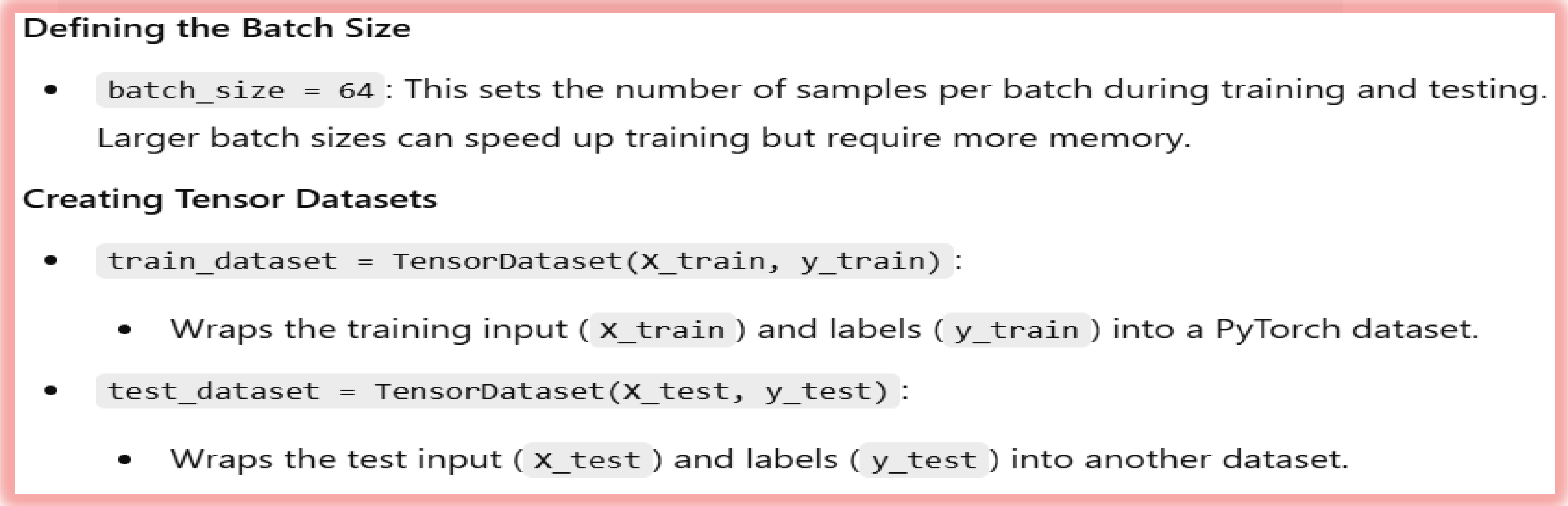
`test_size=0.2` : Specifies that 20% of the dataset should be used for testing, while 80% is used for training.

`random_state=42` : Ensures that the split is reproducible—each time you run the code, you'll get the same training and testing sets.

- Creating DataLoader objects to efficiently handle batching for training and testing in PyTorch.



```python
# Create DataLoader for batching
batch_size = 64
train_dataset = TensorDataset(X_train, y_train)
test_dataset = TensorDataset(X_test, y_test)
```



Defining the Batch Size

- `batch_size = 64`: This sets the number of samples per batch during training and testing. Larger batch sizes can speed up training but require more memory.

Creating Tensor Datasets

- `train_dataset = TensorDataset(X_train, y_train)`:
  - Wraps the training input (`X_train`) and labels (`y_train`) into a PyTorch dataset.
- `test_dataset = TensorDataset(X_test, y_test)`:
  - Wraps the test input (`X_test`) and labels (`y_test`) into another dataset.

```
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size)
```

`train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)`:

- Creates a `DataLoader` for the training dataset.
- `shuffle=True` ensures that the training data is shuffled in every epoch to improve learning and generalization.

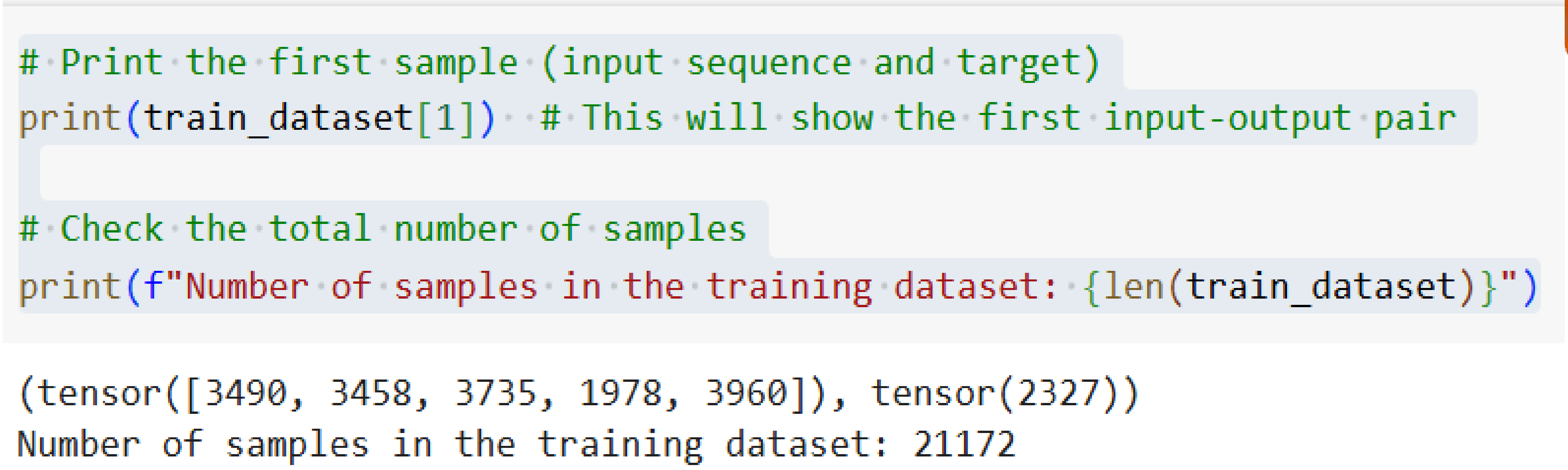
`test_loader = DataLoader(test_dataset, batch_size=batch_size)`:

- Creates a `DataLoader` for the test dataset.
- No `shuffle=True` because test data should be evaluated in a fixed order.

```python
print(train_dataset)
```

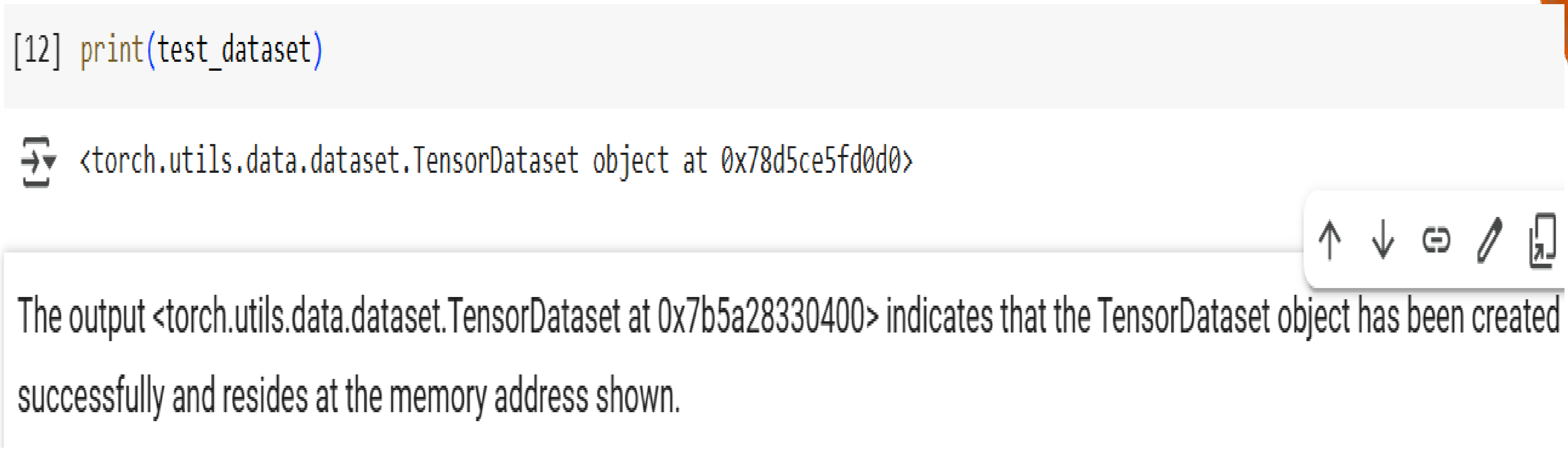
```
<torch.utils.data.dataset.TensorDataset object at 0x78d5c10c26d0>
```

The output <torch.utils.data.dataset.TensorDataset at 0x7b5a28330190> is just Python's way of printing the location in memory of the TensorDataset object. It doesn't represent any issue; it just means you've successfully created a TensorDataset. If you'd like to see the content of the dataset instead, you can inspect it more directly as shown below:



```python
# Print the first sample (input sequence and target)
print(train_dataset[1])  # This will show the first input-output pair

# Check the total number of samples
print(f"Number of samples in the training dataset: {len(train_dataset)}")
```

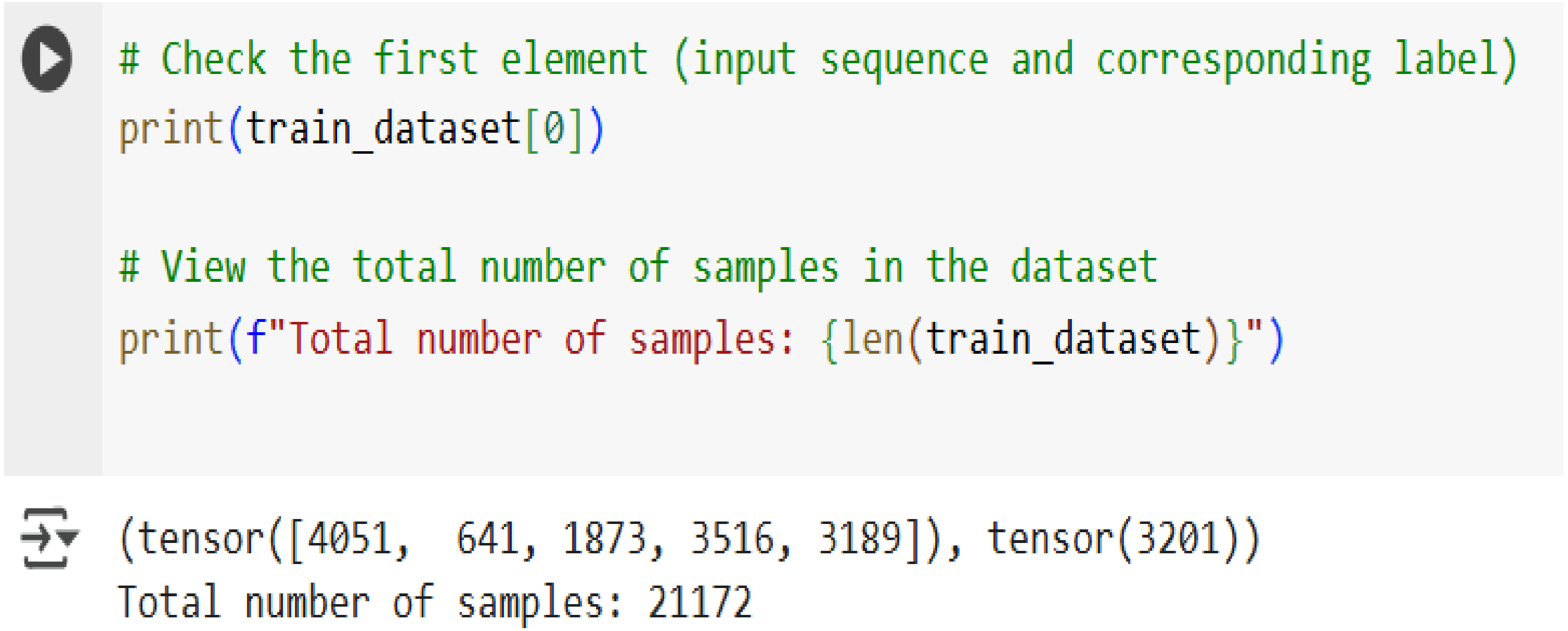
```
(tensor([3490, 3458, 3735, 1978, 3960]), tensor(2327))
Number of samples in the training dataset: 21172
```



```
[12] print(test_dataset)
```

```
<torch.utils.data.dataset.TensorDataset object at 0x78d5ce5fd0d0>
```

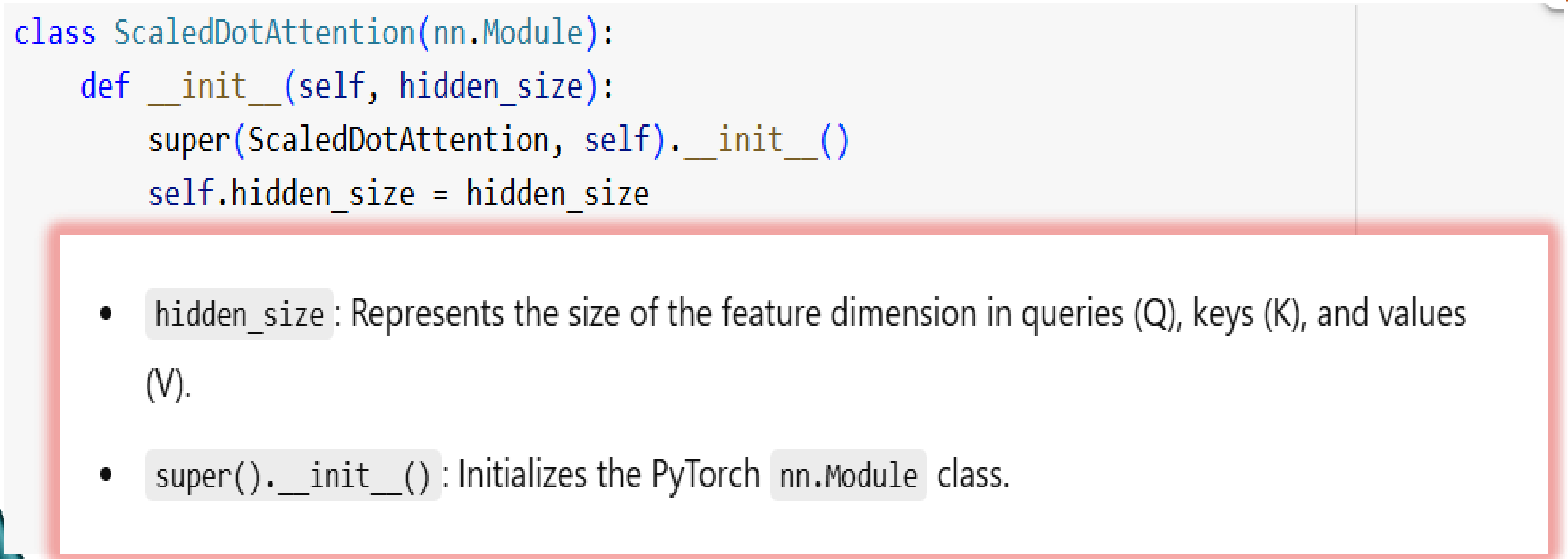
The output <torch.utils.data.dataset.TensorDataset at 0x7b5a28330400> indicates that the TensorDataset object has been created successfully and resides at the memory address shown.



```python
# Check the first element (input sequence and corresponding label)
print(train_dataset[0])

# View the total number of samples in the dataset
print(f"Total number of samples: {len(train_dataset)}")
```

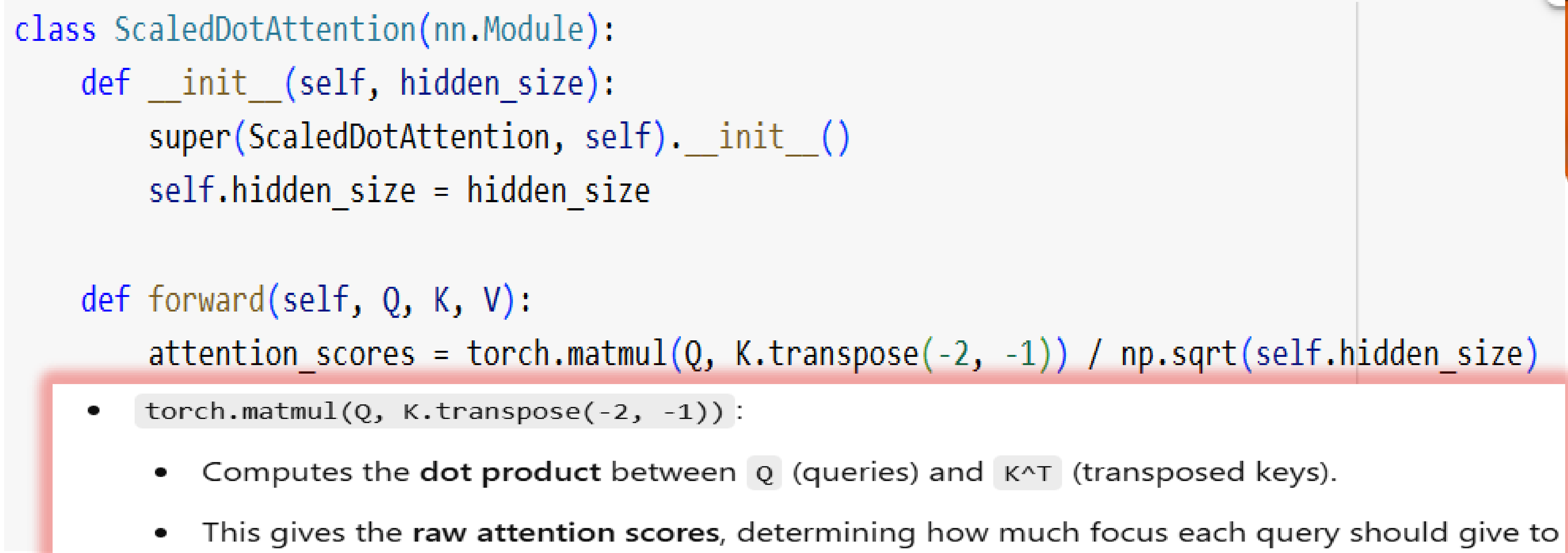
```
(tensor([4051,  641, 1873, 3516, 3189]), tensor(3201))
Total number of samples: 21172
```

Define the Scaled Dot-Product Attention Mechanism

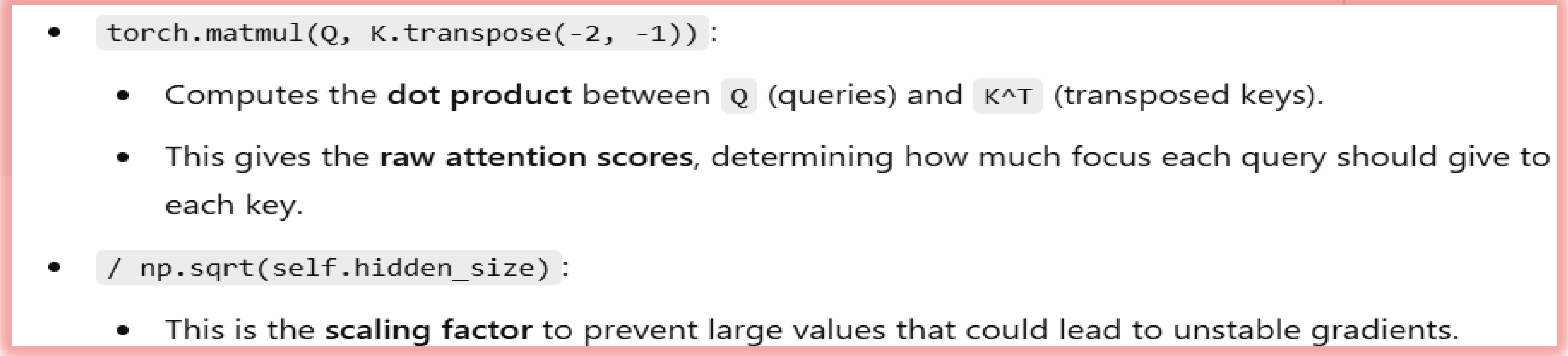


```python
class ScaledDotAttention(nn.Module):
    def __init__(self, hidden_size):
        super(ScaledDotAttention, self).__init__()
        self.hidden_size = hidden_size
```

- `hidden_size`: Represents the size of the feature dimension in queries (Q), keys (K), and values (V).
- `super().__init__()`: Initializes the PyTorch `nn.Module` class.



```python
class ScaledDotAttention(nn.Module):
    def __init__(self, hidden_size):
        super(ScaledDotAttention, self).__init__()
        self.hidden_size = hidden_size

    def forward(self, Q, K, V):
        attention_scores = torch.matmul(Q, K.transpose(-2, -1)) / np.sqrt(self.hidden_size)
```



- `torch.matmul(Q, K.transpose(-2, -1))`:
  - Computes the **dot product** between `Q` (queries) and `K^T` (transposed keys).
  - This gives the **raw attention scores**, determining how much focus each query should give to each key.
- `/ np.sqrt(self.hidden_size)`:
  - This is the **scaling factor** to prevent large values that could lead to unstable gradients.

```
class ScaledDotAttention(nn.Module):
    def __init__(self, hidden_size):
        super(ScaledDotAttention, self).__init__()
        self.hidden_size = hidden_size

    def forward(self, Q, K, V):
        attention_scores = torch.matmul(Q, K.transpose(-2, -1)) / np.sqrt(self.hidden_size)
        attention_weights = torch.softmax(attention_scores, dim=-1)
```

- Converts raw attention scores into **probabilities** (attention distribution).
- Ensures that each row of attention scores sums to 1 (helping interpretability).

```
class ScaledDotAttention(nn.Module):
    def __init__(self, hidden_size):
        super(ScaledDotAttention, self).__init__()
        self.hidden_size = hidden_size

    def forward(self, Q, K, V):
        attention_scores = torch.matmul(Q, K.transpose(-2, -1)) / np.sqrt(self.hidden_size)
        attention_weights = torch.softmax(attention_scores, dim=-1)
        output = torch.matmul(attention_weights, V)
```

`torch.matmul(attention_weights, V)`:

- Applies the attention weights to the values `V`, producing the final output.
- This step determines how much of each value contributes to the final representation.

This module takes in three tensors:

- **Q (Query)** → What we want to find relevant information for.
- **K (Key)** → The reference for comparison.
- **V (Value)** → The actual data that will be weighted based on attention scores.

The output is a **weighted sum** of values `V`, where the attention scores determine how much each value contributes.

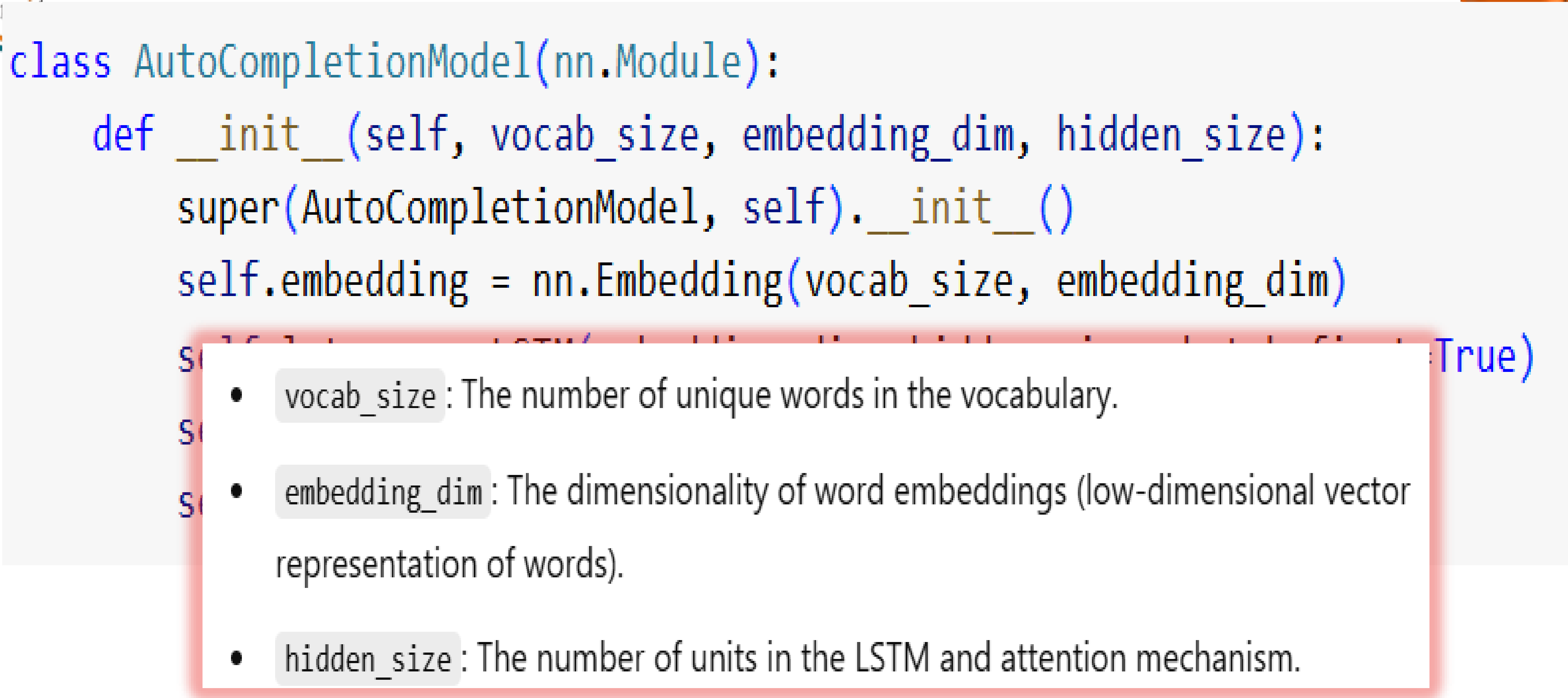
Possible Improvements

1. **Masking Support**
   - To handle **padding** in NLP tasks, you might need an **attention mask** to ignore certain tokens.
2. **Dropout**
   - Attention mechanisms often use `nn.Dropout` for regularization.
3. **Multi-Head Attention Extension**
   - In Transformers, **Multi-Head Attention** extends this by using multiple attention heads with different learned projections.

Define the Auto-Completion Model

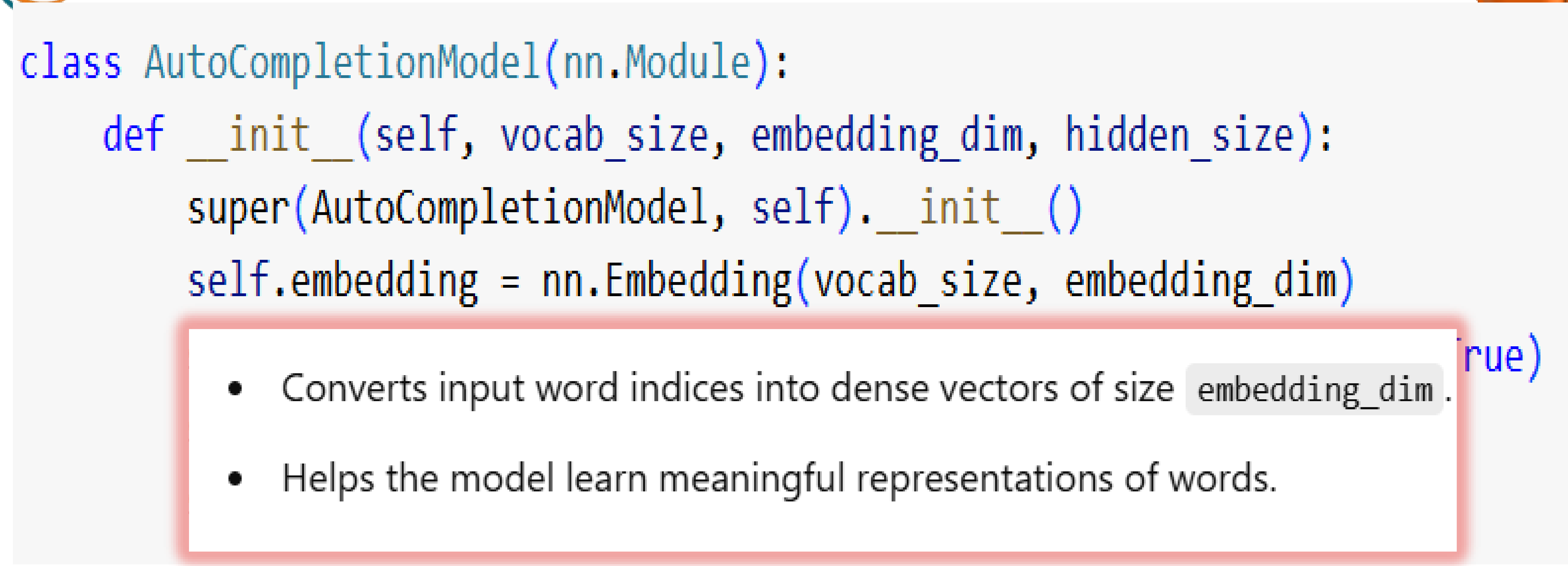
- AutoCompletionModel is a sequence-to-sequence model with attention, designed for predicting the next word in a text sequence.
- It combines an embedding layer, an LSTM, a scaled dot-product attention mechanism, and a fully connected layer for output.

```python
class AutoCompletionModel(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_size):
        super(AutoCompletionModel, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_size, batch_first=True)
        self.attention = ScaledDotAttention(hidden_size)
        self.fc = nn.Linear(hidden_size, vocab_size)

    def forward(self, x):
        embedded = self.embedding(x)  # Embedding layer
        lstm_out, _ = self.lstm(embedded)  # LSTM layer
        attention_out = self.attention(lstm_out, lstm_out, lstm_out)  # Attention mechanism
        out = self.fc(attention_out[:, -1, :])  # Fully connected layer
        return out
```



```
class AutoCompletionModel(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_size):
        super(AutoCompletionModel, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        s                                                      True)
        s
        s
```

- `vocab_size`: The number of unique words in the vocabulary.
- `embedding_dim`: The dimensionality of word embeddings (low-dimensional vector representation of words).
- `hidden_size`: The number of units in the LSTM and attention mechanism.



```python
class AutoCompletionModel(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_size):
        super(AutoCompletionModel, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
                                                                    rue)
```

- Converts input word indices into dense vectors of size `embedding_dim`.
- Helps the model learn meaningful representations of words.

```
class AutoCompletionModel(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_size):
        super(AutoCompletionModel, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_size, batch_first=True)
        s
        s
```

- Processes the sequence word by word and captures **long-term dependencies**.
- Outputs a sequence of **hidden states** for each word.

Why LSTM?

- It avoids the vanishing gradient problem of simple RNNs.
- Suitable for sequential data like text.

```
class AutoCompletionModel(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_size):
        super(AutoCompletionModel, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_size, batch_first=True)
        self.attention = ScaledDotAttention(hidden_size)
```

- The LSTM output (`lstm_out`) serves as **queries (Q)**, **keys (K)**, **and values (V)**.
- Helps the model focus on relevant words from previous timesteps while generating the next word.

```
class AutoCompletionModel(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_size):
        super(AutoCompletionModel, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_size, batch_first=True)
        self.attention = ScaledDotAttention(hidden_size)
        self.fc = nn.Linear(hidden_size, vocab_size)
```

- Maps the attention output to a vector of **vocab_size**, representing probabilities for the next word.

```python
def forward(self, x):
    embedded = self.embedding(x)  # Embedding layer
    lstm_out, _ = self.lstm(embedded)  # LSTM layer
    attention_out = self.attention(lstm_out, lstm_out, lstm_out)  # Attention mechanism
    out = self.fc(attention_out[:, -1, :])  # Fully connected layer
    return out
```

```
embedded = self.embedding(x)
```

- Converts input token indices into embedding vectors.

```
lstm_out, _ = self.lstm(embedded)
```

- Processes the sequence and outputs hidden states.

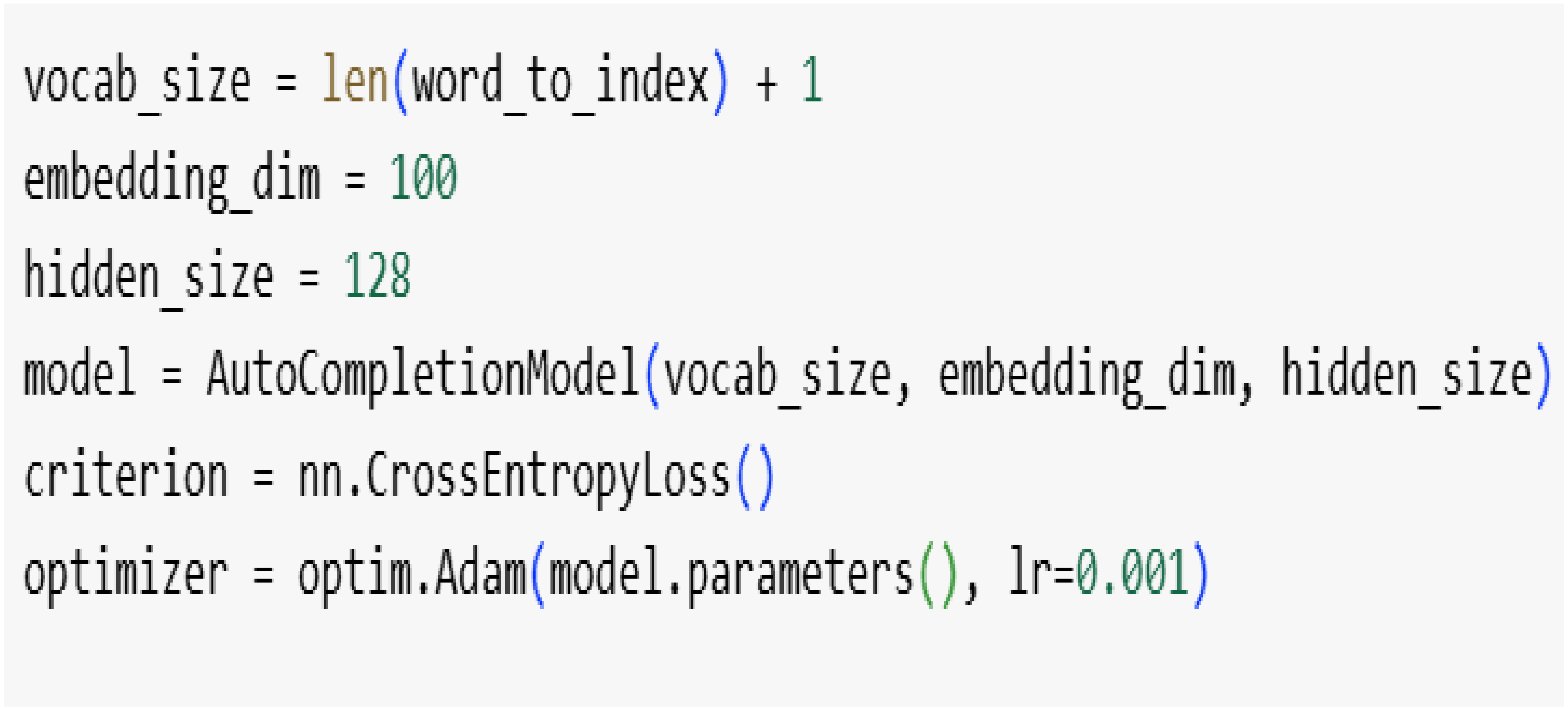
```
attention_out = self.attention(lstm_out, lstm_out, lstm_out)
```

- Applies **self-attention** to focus on important words.

```
out = self.fc(attention_out[:, -1, :])
```

- Extracts the last time step's attention output.
- Passes it through `fc` to produce **vocabulary-sized logits** (predictions for the next word).

Instantiate Model, Loss, and Optimizer

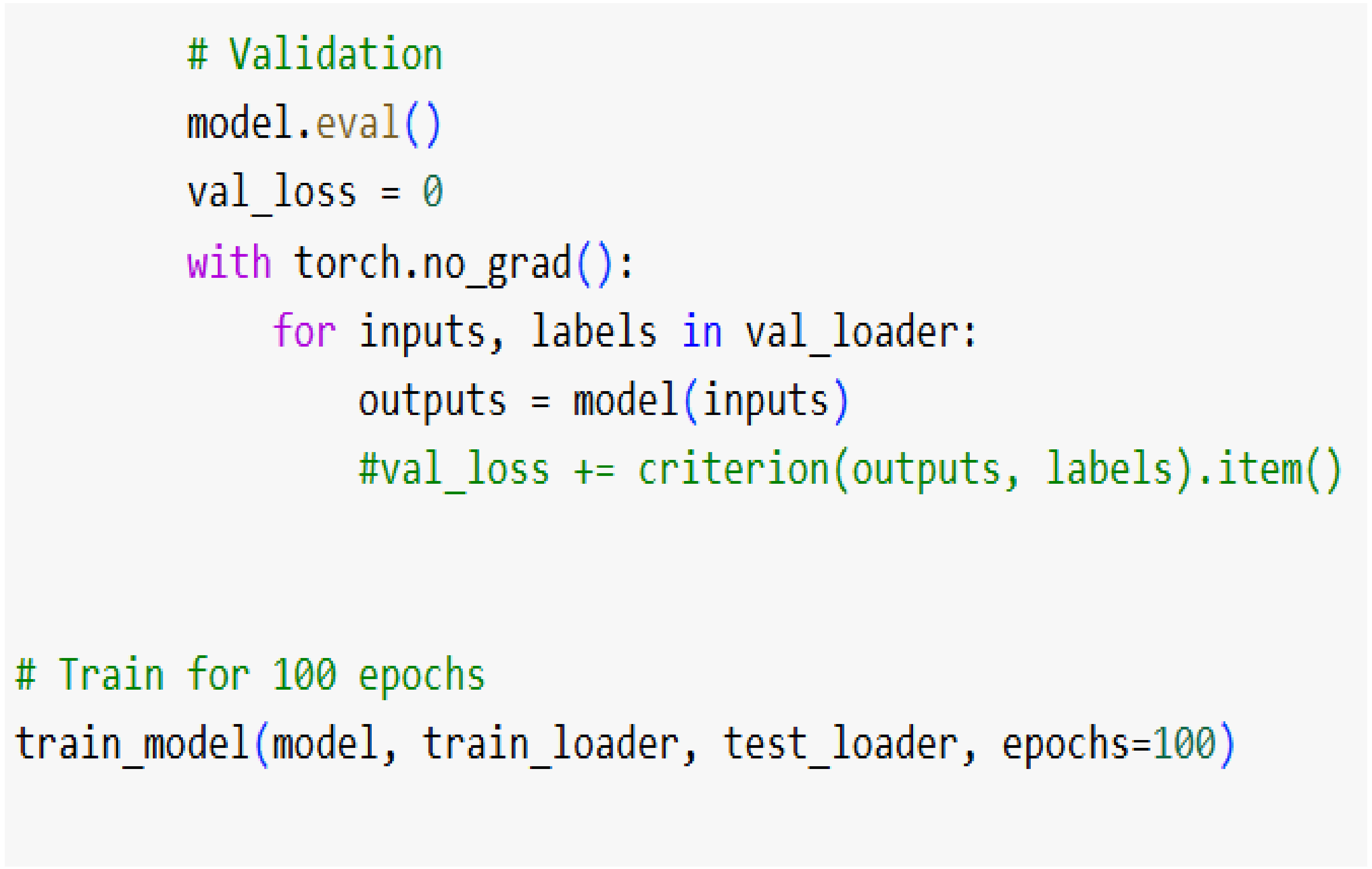


```
vocab_size = len(word_to_index) + 1
embedding_dim = 100
hidden_size = 128
model = AutoCompletionModel(vocab_size, embedding_dim, hidden_size)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

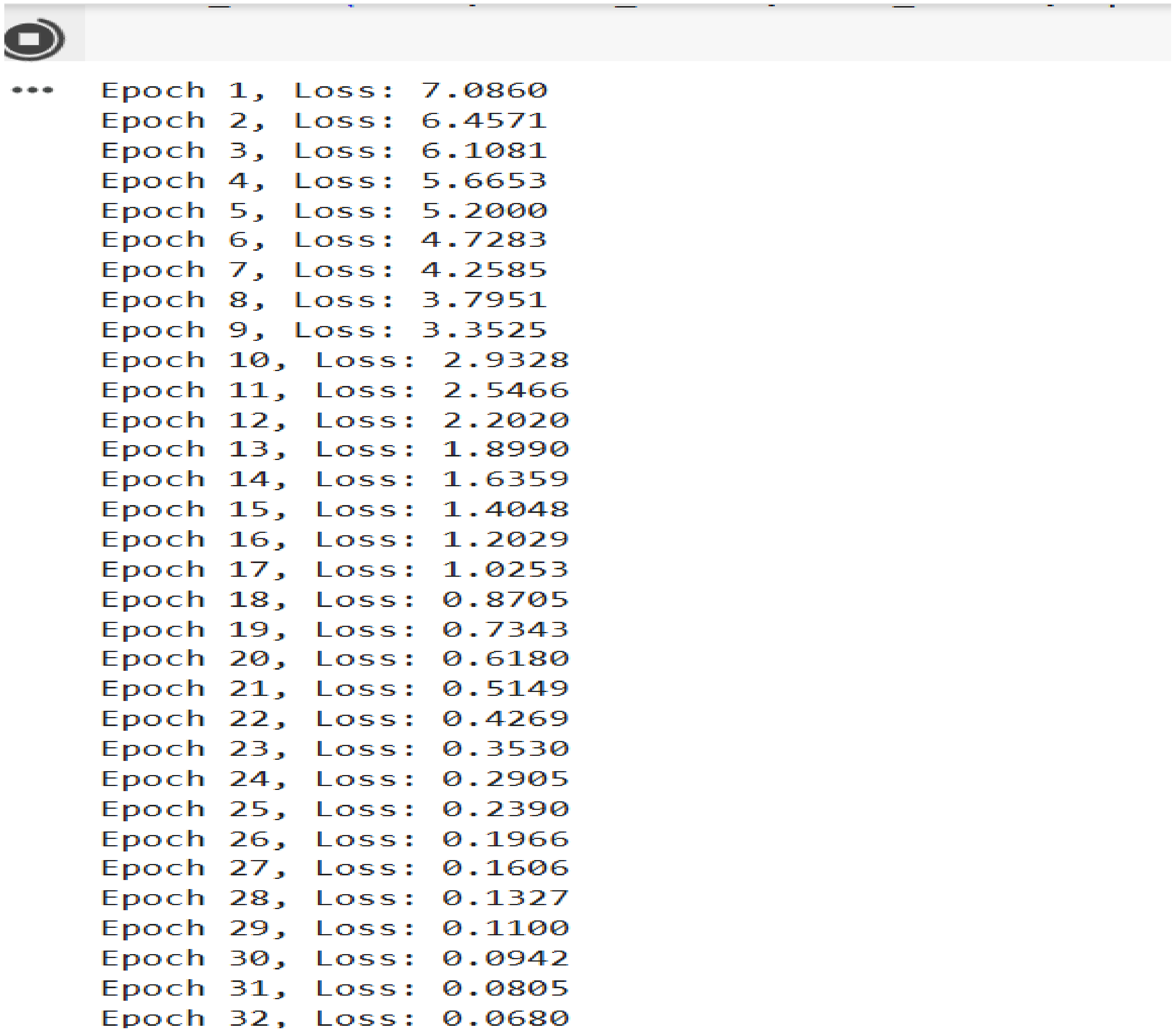
Train the Model

```python
def train_model(model, train_loader, val_loader, epochs):
    for epoch in range(epochs):
        model.train()
        total_loss = 0

        for inputs, labels in train_loader:
            outputs = model(inputs)
            loss = criterion(outputs, labels)

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            total_loss += loss.item()

        avg_loss = total_loss / len(train_loader)
        print(f"Epoch {epoch+1}, Loss: {avg_loss:.4f}")

        # Validation
        model.eval()
        val_loss = 0
        with torch.no_grad():
            for inputs, labels in val_loader:
                outputs = model(inputs)
                #val_loss += criterion(outputs, labels).item()
        #print(f"Validation Loss: {val_loss / len(val_loader):.4f}")

# Train for 100 epochs
train_model(model, train_loader, test_loader, epochs=100)
```

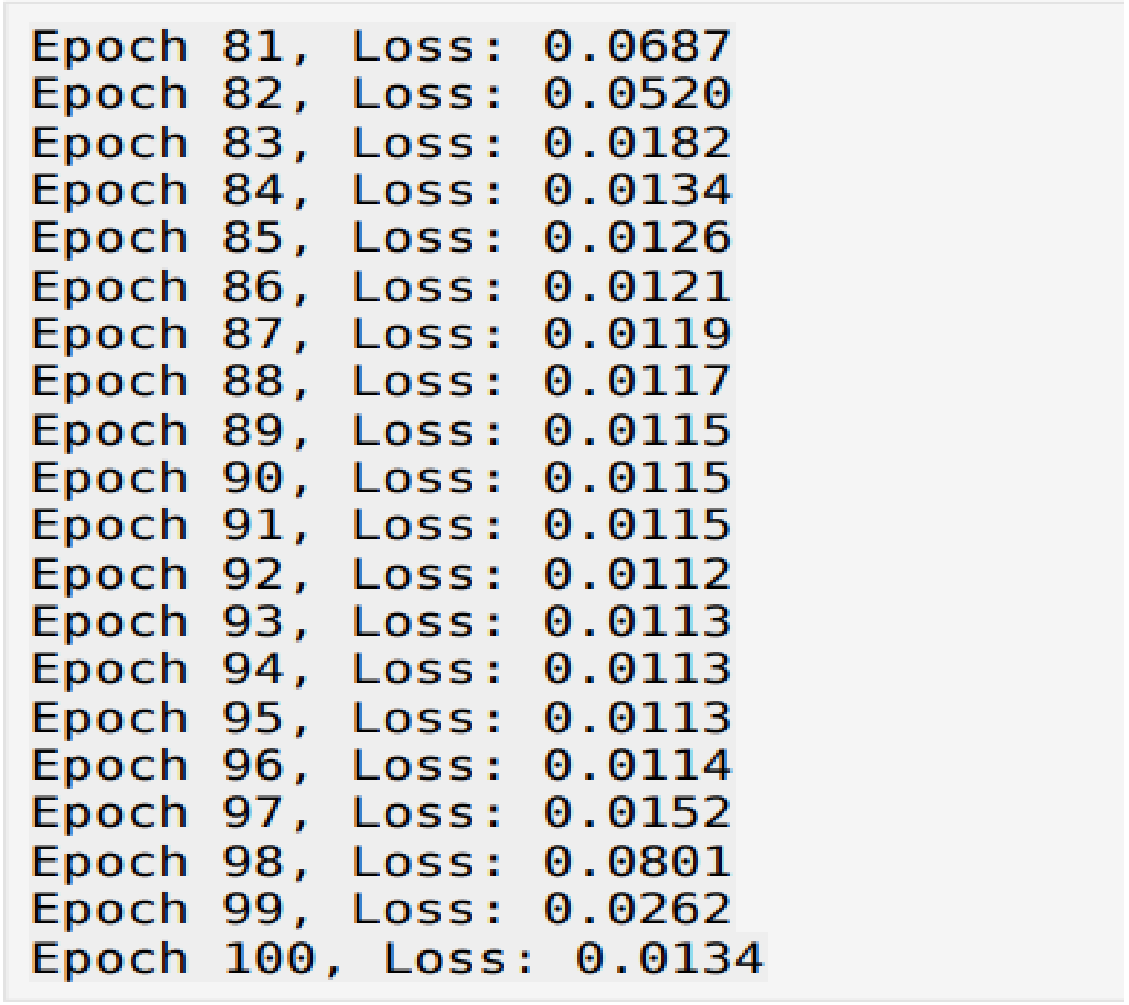
```python
def train_model(model, train_loader, val_loader, epochs):
    for epoch in range(epochs):
        model.train()
        total_loss = 0

        for inputs, labels in train_loader:
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            total_loss += loss.item()

        avg_loss = total_loss / len(train_loader)
        print(f"Epoch {epoch+1}, Loss: {avg_loss:.4f}")
```



```python
        # Validation
        model.eval()
        val_loss = 0
        with torch.no_grad():
            for inputs, labels in val_loader:
                outputs = model(inputs)
                #val_loss += criterion(outputs, labels).item()



# Train for 100 epochs
train_model(model, train_loader, test_loader, epochs=100)
```

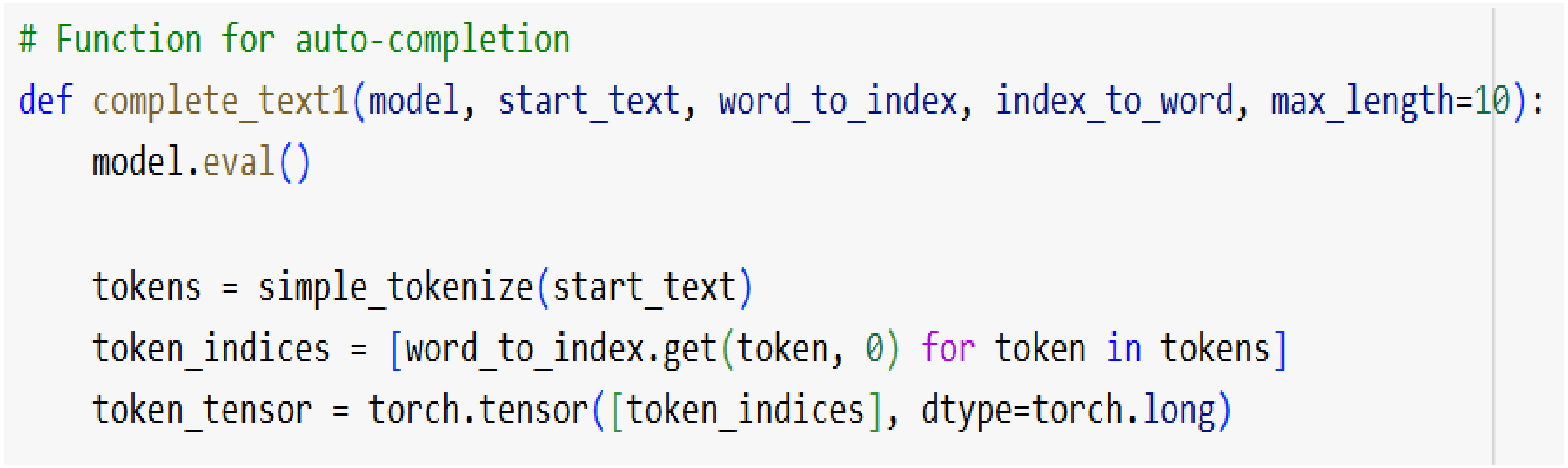


```
Epoch 1, Loss: 7.0860
Epoch 2, Loss: 6.4571
Epoch 3, Loss: 6.1081
Epoch 4, Loss: 5.6653
Epoch 5, Loss: 5.2000
Epoch 6, Loss: 4.7283
Epoch 7, Loss: 4.2585
Epoch 8, Loss: 3.7951
Epoch 9, Loss: 3.3525
Epoch 10, Loss: 2.9328
Epoch 11, Loss: 2.5466
Epoch 12, Loss: 2.2020
Epoch 13, Loss: 1.8990
Epoch 14, Loss: 1.6359
Epoch 15, Loss: 1.4048
Epoch 16, Loss: 1.2029
Epoch 17, Loss: 1.0253
Epoch 18, Loss: 0.8705
Epoch 19, Loss: 0.7343
Epoch 20, Loss: 0.6180
Epoch 21, Loss: 0.5149
Epoch 22, Loss: 0.4269
Epoch 23, Loss: 0.3530
Epoch 24, Loss: 0.2905
Epoch 25, Loss: 0.2390
Epoch 26, Loss: 0.1966
Epoch 27, Loss: 0.1606
Epoch 28, Loss: 0.1327
Epoch 29, Loss: 0.1100
Epoch 30, Loss: 0.0942
Epoch 31, Loss: 0.0805
Epoch 32, Loss: 0.0680
```



```
Epoch 81, Loss: 0.0687
Epoch 82, Loss: 0.0520
Epoch 83, Loss: 0.0182
Epoch 84, Loss: 0.0134
Epoch 85, Loss: 0.0126
Epoch 86, Loss: 0.0121
Epoch 87, Loss: 0.0119
Epoch 88, Loss: 0.0117
Epoch 89, Loss: 0.0115
Epoch 90, Loss: 0.0115
Epoch 91, Loss: 0.0115
Epoch 92, Loss: 0.0112
Epoch 93, Loss: 0.0113
Epoch 94, Loss: 0.0113
Epoch 95, Loss: 0.0113
Epoch 96, Loss: 0.0114
Epoch 97, Loss: 0.0152
Epoch 98, Loss: 0.0801
Epoch 99, Loss: 0.0262
Epoch 100, Loss: 0.0134
```

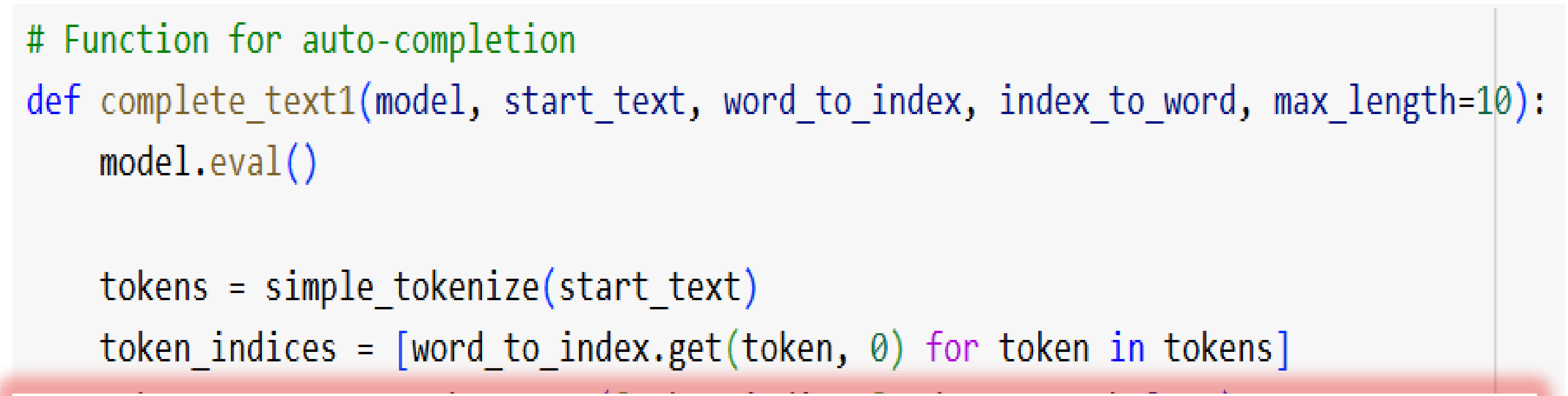
Text auto completion



```
# Function for auto-completion
def complete_text1(model, start_text, word_to_index, index_to_word, max_length=10):
    model.eval()

    tokens = simple_tokenize(start_text)
    token_indices = [word_to_index.get(token, 0) for token in tokens]
    token_tensor = torch.tensor([token_indices], dtype=torch.long)
```

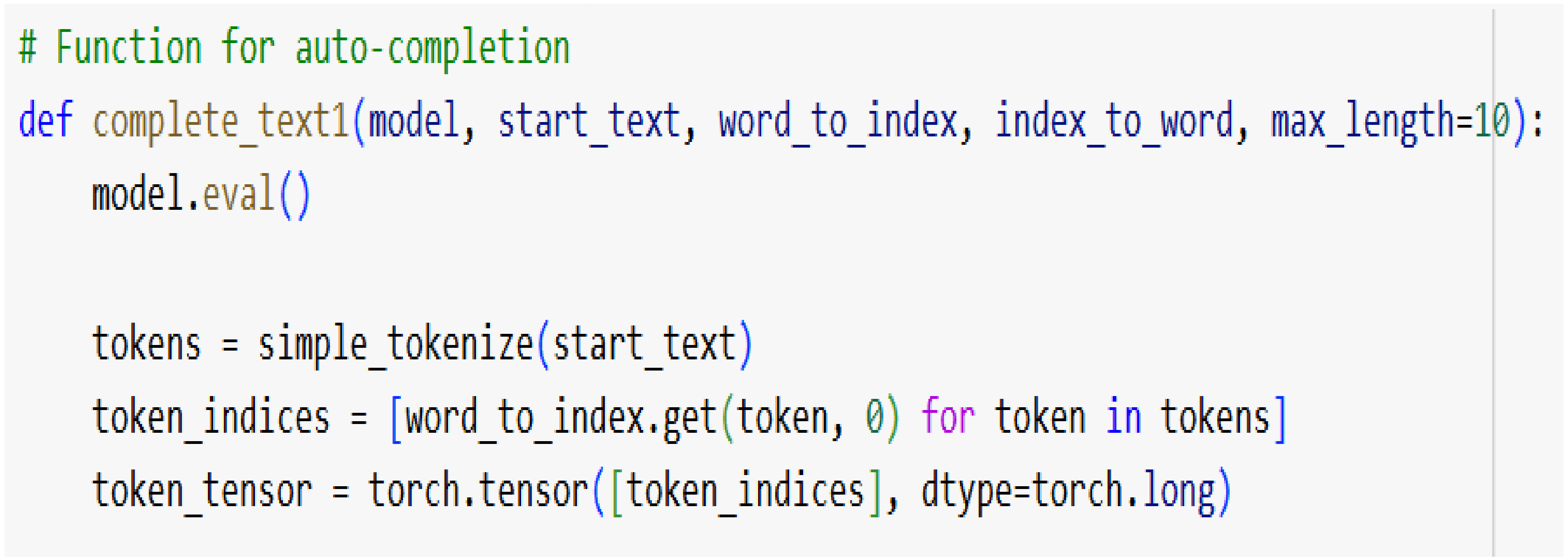
```python
# Function for auto-completion
def complete_text1(model, start_text, word_to_index, index_to_word, max_length=10):
    model.eval()

    tokens = simple_tokenize(start_text)
    token_indices = [word_to_index.get(token, 0) for token in tokens]
    token_tensor = torch.tensor([token_indices], dtype=torch.long)
```

- `model` : The trained model (e.g., `AutoCompletionModel`) that will generate predictions.
- `start_text` : The initial text you want to use as the prompt for generating the rest of the text.
- `word_to_index` : A dictionary that maps words to their corresponding indices.
- `index_to_word` : A dictionary that maps indices back to their corresponding words.
- `max_length=10` : The maximum number of words to generate. This controls how long the auto-completed text will be.

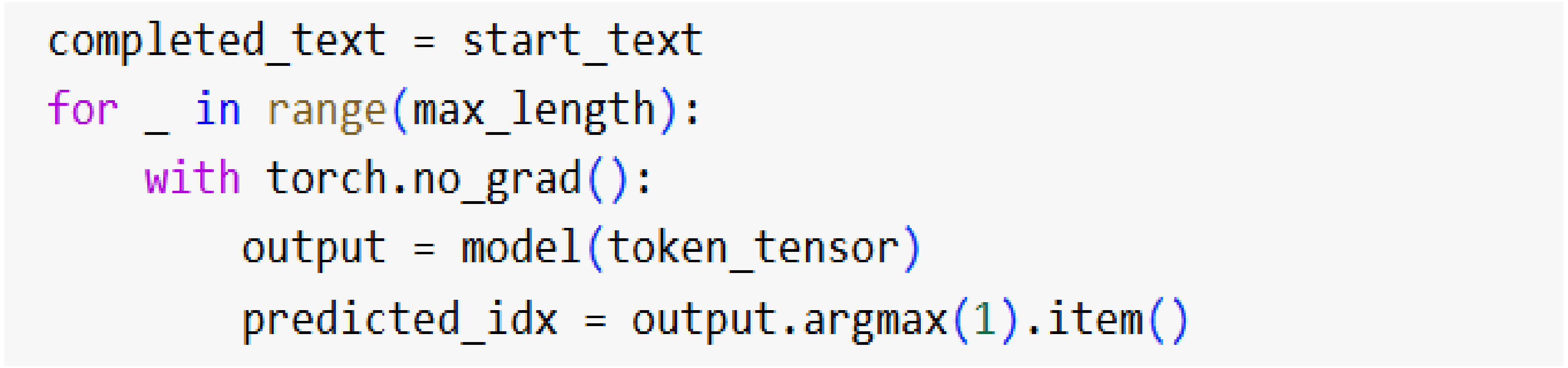


```python
# Function for auto-completion
def complete_text1(model, start_text, word_to_index, index_to_word, max_length=10):
    model.eval()

    tokens = simple_tokenize(start_text)
    token_indices = [word_to_index.get(token, 0) for token in tokens]
```

- `simple_tokenize(start_text)` tokenizes the `start_text` into a list of words (lowercased).
- `word_to_index.get(token, 0)` looks up the index of each token (word) in the `word_to_index` dictionary. If the word is not found, it returns `0` (which can be treated as a padding token or unknown word index).



```
# Function for auto-completion
def complete_text1(model, start_text, word_to_index, index_to_word, max_length=10):
    model.eval()

    tokens = simple_tokenize(start_text)
    token_indices = [word_to_index.get(token, 0) for token in tokens]
    token_tensor = torch.tensor([token_indices], dtype=torch.long)
```



```
    completed_text = start_text
    for _ in range(max_length):
        with torch.no_grad():
            output = model(token_tensor)
            predicted_idx = output.argmax(1).item()
```

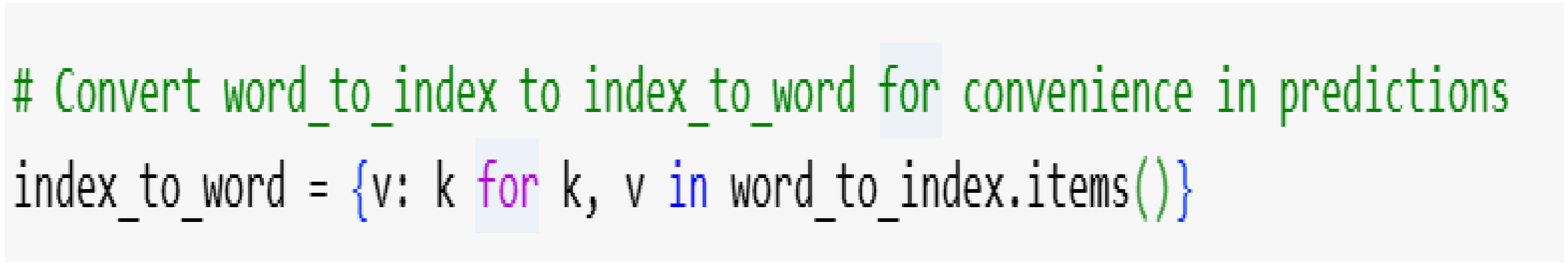
```
completed_text = start_text
for _ in range(max_length):
    with torch.no_grad():
        output = model(token_tensor)
        predicted_idx = output.argmax(1).item()
```

Looping for `max_length`: This loop will run up to `max_length` times, generating one word at a time.

- `with torch.no_grad()`: This ensures that gradients are not calculated during inference, saving memory and computation.
- `output = model(token_tensor)`: The model takes the `token_tensor` (which is the list of indices representing the current sequence) and outputs a prediction for the next word.
- `predicted_idx = output.argmax(1).item()`: The model's output is a probability distribution across the vocabulary. `argmax(1)` finds the index with the highest probability (i.e., the predicted next word). `.item()` retrieves the scalar value of the index.

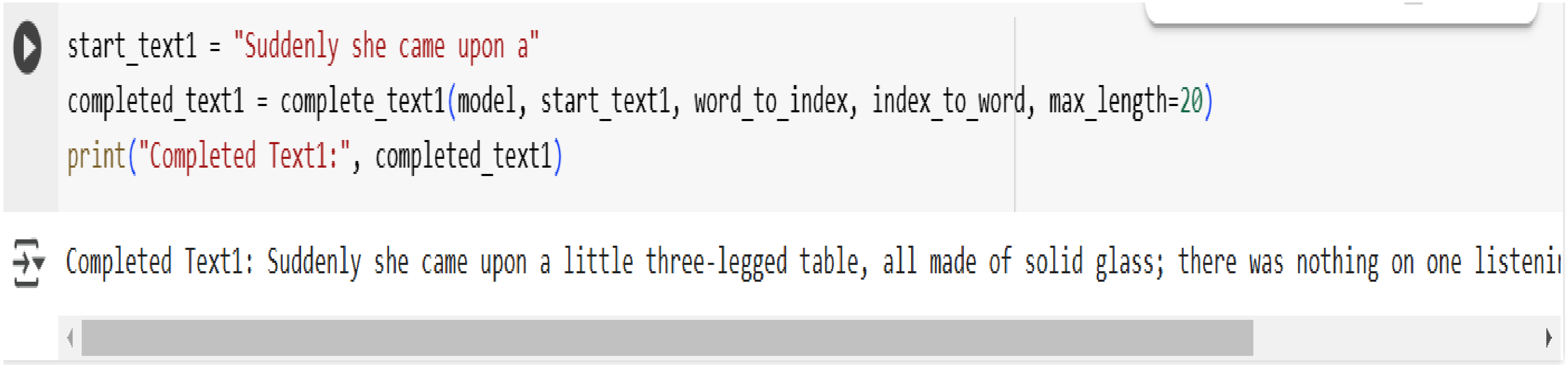
```python
        predicted_word = index_to_word[predicted_idx]
        completed_text += ' ' + predicted_word

        # Update token_tensor to include the new word
        token_indices.append(predicted_idx)
        token_tensor = torch.tensor([token_indices[-sequence_length:]], dtype=torch.long)

    return completed_text
```

- `predicted_word = index_to_word[predicted_idx]`: The predicted index is mapped back the word using `index_to_word`.
- `completed_text += ' ' + predicted_word`: The predicted word is added to the `completed_text`.
- **Update the input sequence for the next prediction**:
  - `token_indices.append(predicted_idx)` adds the predicted word's index to the sequence.
  - `token_tensor = torch.tensor([token_indices[-sequence_length:]], dtype=torch.long)` creates a new tensor by keeping only the last `sequence_length` words, ensuring that the sequence fed to the model is always of the same length.



```python
# Convert word_to_index to index_to_word for convenience in predictions
index_to_word = {v: k for k, v in word_to_index.items()}
```

Testing



```
start_text1 = "Suddenly she came upon a"
completed_text1 = complete_text1(model, start_text1, word_to_index, index_to_word, max_length=20)
print("Completed Text1:", completed_text1)
```

```
Completed Text1: Suddenly she came upon a little three-legged table, all made of solid glass; there was nothing on one listeni
```

```
start_text1 = "Suddenly she came upon a"
completed_text1 = complete_text1(model, start_text1, word_to_index,
index_to_word, max_length=20)
print("Completed Text1:", completed_text1)
```

```
Completed Text1: Suddenly she came upon a little three-legged table,
all made of solid glass; there was nothing on to do, one of a little
golden and
```

```
start_text2 = "the white rabbit put on his spectacles"
completed_text2 = complete_text1(model, start_text2, word_to_index,
index_to_word, max_length=20)
print("Completed Text2:", completed_text2)
```

```
Completed Text2: the white rabbit put on his spectacles and looked at
round, to see the executioner was going a large ring, with the air. in
the duchess sneezed
```

```
start_text3 = "The rabbit-hole went straight on like"
completed_text3 = complete_text1(model, start_text3, word_to_index,
index_to_word, max_length=20)
print("Completed Text3:", completed_text3)
```

```
Completed Text3: The rabbit-hole went straight on like like a tunnel
with it turned and the table to get with its arms folded, by round a
little ledge
```

Thank you!