NLP PROJECT

Team: Hats

Team Members:

Harsh Kumar: 21ucs086
Om Gupta: 21ucs140
Pitani Nitin Srichakri: 21ucs149
Tarun Agarwal: 21ucs219

Book: “The White Company”- Arthur Conan Doyle

Submitted to: Dr. Shakti Balan

ACKNOWLEDGEMENT

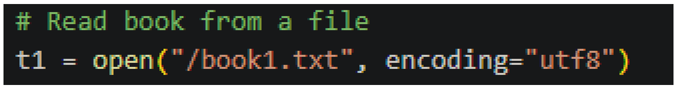
We are grateful to our respected teacher, Dr. Sakthi Balan, whose provided knowledge in classes benefited us in completing this project successfully. Thank you so much for your continuous support and presence whenever needed. Lastly, we would like to thank everyone involved in the project directly or indirectly. And we are always thankful to The Great Almighty for always having a blessing on us

AIM:

This project aims to obtain PoS tagging on a Novel, calculate bigram probability, and play the Shannon game.

Data Description:

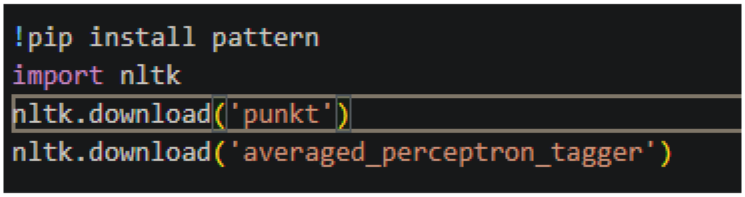
The novel we use for this project is “ The white company”- Arthur Canon Doyle. Here is complete link to the project.



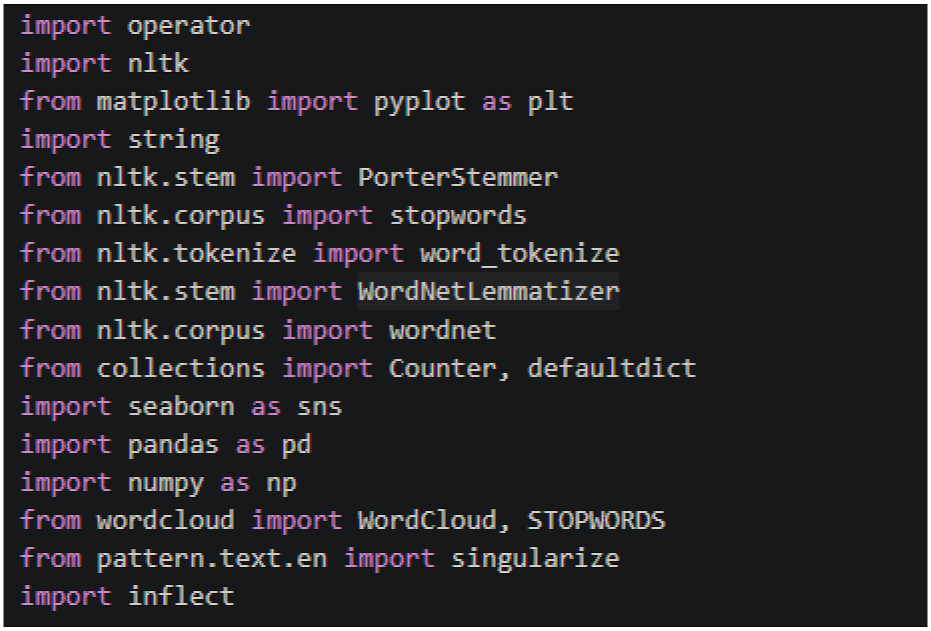
```
# Read book from a file
t1 = open("/book1.txt", encoding="utf8")
```

Firstly, importing some libraries for various tasks

- Importing the ‘pandas’ module for data analysis and machine learning tasks.
- Importing ‘nltk’(natural language toolkit) for the NLP-related tasks.
- Importing ‘matplotlib.pyplot’ and “Seaborn” for graphs and frequency analysis.



```
!pip install pattern
import nltk
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')
```

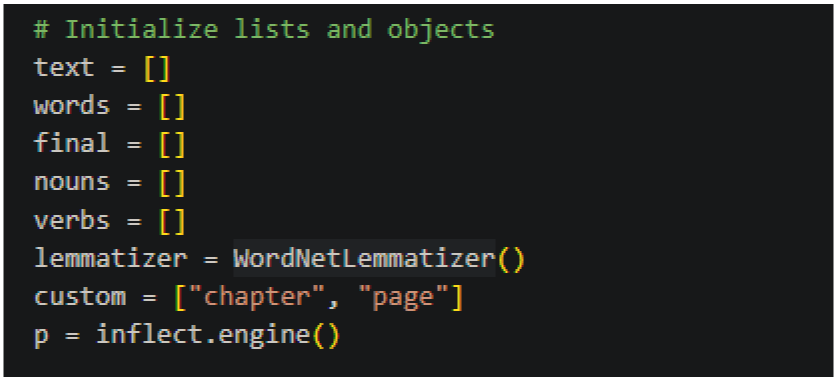


```
import operator
import nltk
from matplotlib import pyplot as plt
import string
from nltk.stem import PorterStemmer
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer
from nltk.corpus import wordnet
from collections import Counter, defaultdict
import seaborn as sns
import pandas as pd
import numpy as np
from wordcloud import WordCloud, STOPWORDS
from pattern.text.en import singularize
import inflect
```

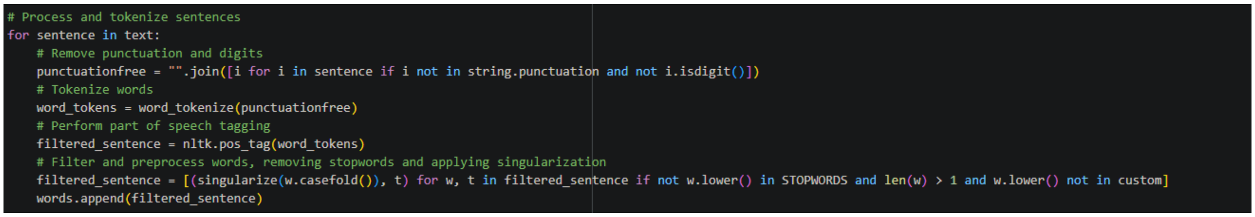
Initialization and Processing:

This code initializes empty lists ('text', 'words', 'final', 'nouns', 'verbs') for storing text data and linguistic elements, along with objects like a lemmatizer ('lemmatizer') and an inflect engine ('p').

- Tokenization and Preprocessing: It processes a list of sentences in the 'text' variable.
- Cleaning: Removes punctuation and digits from the sentence.
- Tokenization: Breaks the cleaned sentence into individual words.
- Part of Speech Tagging: Tags each word with its part of speech.
- Filtering and Preprocessing:
  - Filters and preprocesses words by singularizing them, removing stopwords, ignoring short words, and checking against a custom list.
  - The filtered words and their part-of-speech tags are added to the 'words' list.



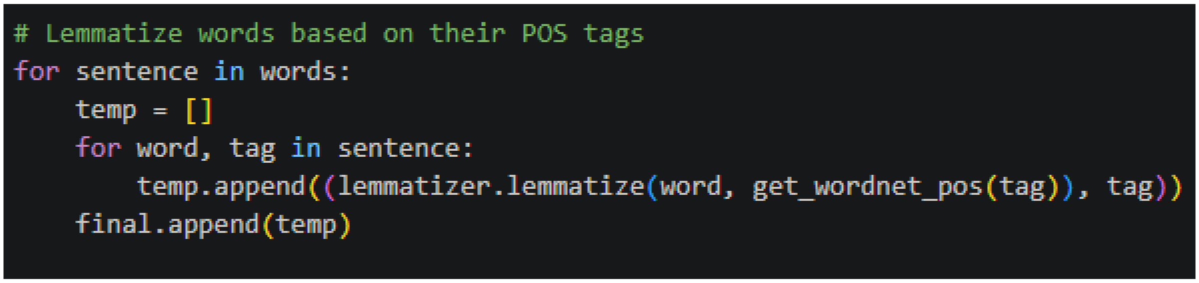
```
# Initialize lists and objects
text = []
words = []
final = []
nouns = []
verbs = []
lemmatizer = WordNetLemmatizer()
custom = ["chapter", "page"]
p = inflect.engine()
```



```
# Process and tokenize sentences
for sentence in text:
    # Remove punctuation and digits
    punctuationfree = "".join([i for i in sentence if i not in string.punctuation and not i.isdigit()])
    # Tokenize words
    word_tokens = word_tokenize(punctuationfree)
    # Perform part of speech tagging
    filtered_sentence = nltk.pos_tag(word_tokens)
    # Filter and preprocess words, removing stopwords and applying singularization
    filtered_sentence = [(singularize(w.casefold()), t) for w, t in filtered_sentence if not w.lower() in STOPWORDS and len(w) > 1 and w.lower() not in custom]
    words.append(filtered_sentence)
```

Lemmatization:

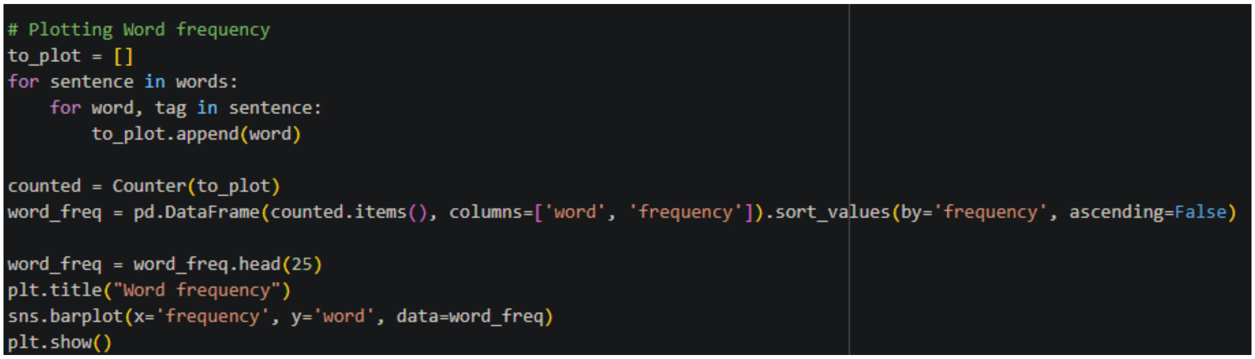
This code lemmatizes words in a list of sentences based on their part-of-speech (POS) tags and stores the lemmatized words and their original POS tags in the 'final' list.



```
# Lemmatize words based on their POS tags
for sentence in words:
    temp = []
    for word, tag in sentence:
        temp.append((lemmatizer.lemmatize(word, get_wordnet_pos(tag)), tag))
    final.append(temp)
```

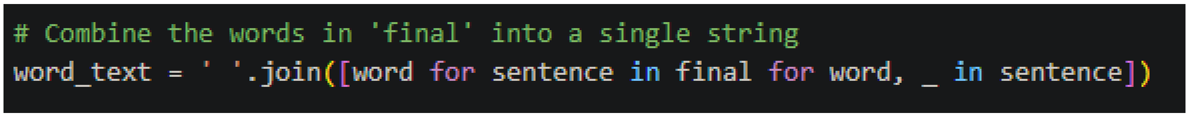
Plotting Frequencies:

- This plots the word frequency of the words in a list of preprocessed sentences ('words').
    - It creates a bar plot showing the top 25 most frequently occurring words, with words on the y-axis and their frequencies on the x-axis.
    - The code uses Python libraries like 'Counter', 'pandas', and 'seaborn' for data manipulation and visualization.



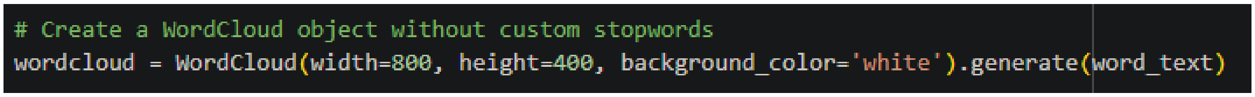
```
# Plotting Word frequency
to_plot = []
for sentence in words:
    for word, tag in sentence:
        to_plot.append(word)

counted = Counter(to_plot)
word_freq = pd.DataFrame(counted.items(), columns=['word', 'frequency']).sort_values(by='frequency', ascending=False)

word_freq = word_freq.head(25)
plt.title("Word frequency")
sns.barplot(x='frequency', y='word', data=word_freq)
plt.show()
```

- Combining words from the 'final' list into a single string called 'word_text' by joining them with spaces creates a continuous text.



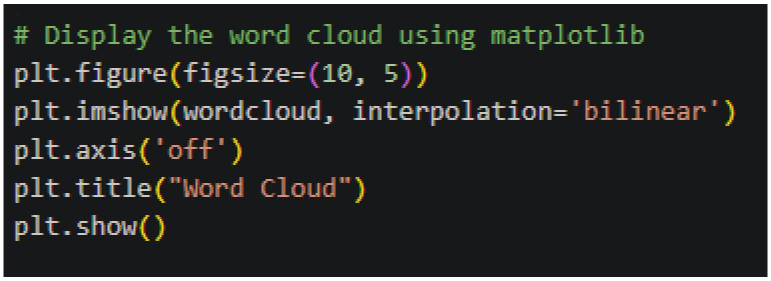
```
# Combine the words in 'final' into a single string
word_text = ' '.join([word for sentence in final for word, _ in sentence])
```

- Creating a WordCloud visualization from the 'word_text' string, where words are represented graphically, without using any custom stopwords.
    - The WordCloud has specific dimensions (800x400 pixels) and a white background.



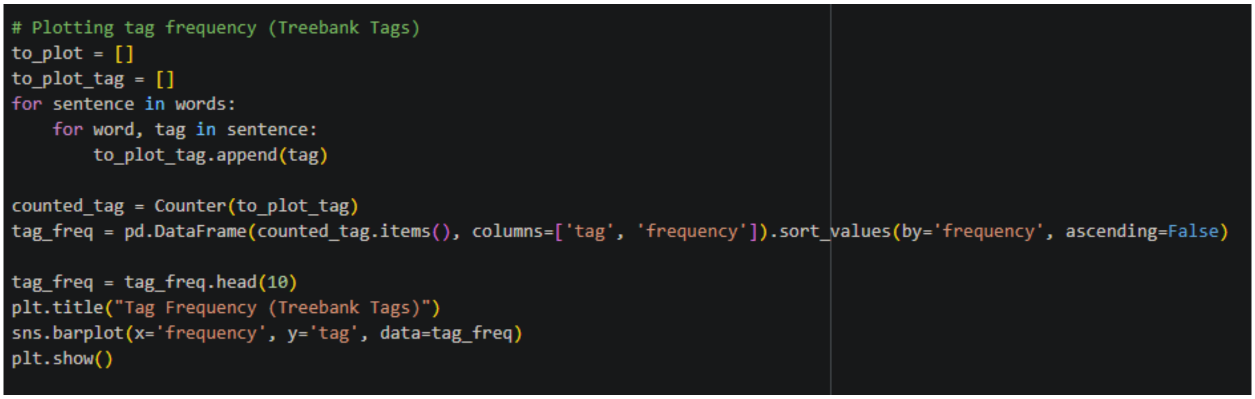
```python
# Create a WordCloud object without custom stopwords
wordcloud = WordCloud(width=800, height=400, background_color='white').generate(word_text)
```

- Displaying a Word Cloud visualization using Matplotlib.
    - It sets the size of the figure, shows the Word Cloud image with a specified interpolation method, removes the axis, sets a title for the plot, and finally displays the Word Cloud.



```python
# Display the word cloud using matplotlib
plt.figure(figsize=(10, 5))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')
plt.title("Word Cloud")
plt.show()
```

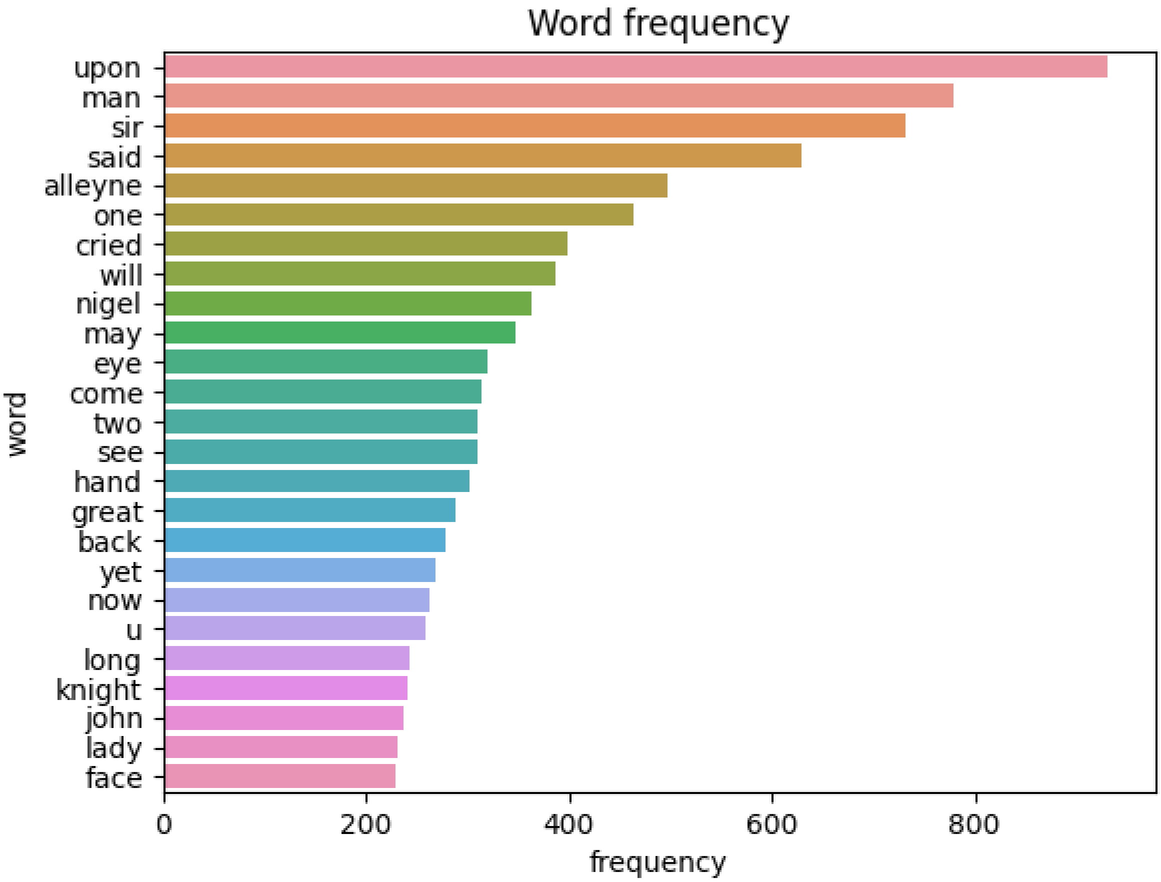
- Plotting the frequency of Treebank tags for words in the 'words' dataset:
    - It collects Treebank tags for each word in the sentences.
    - Counts the frequency of each tag and creates a sorted DataFrame.
    - Displays the top 10 most frequent tags in a bar plot using Seaborn.



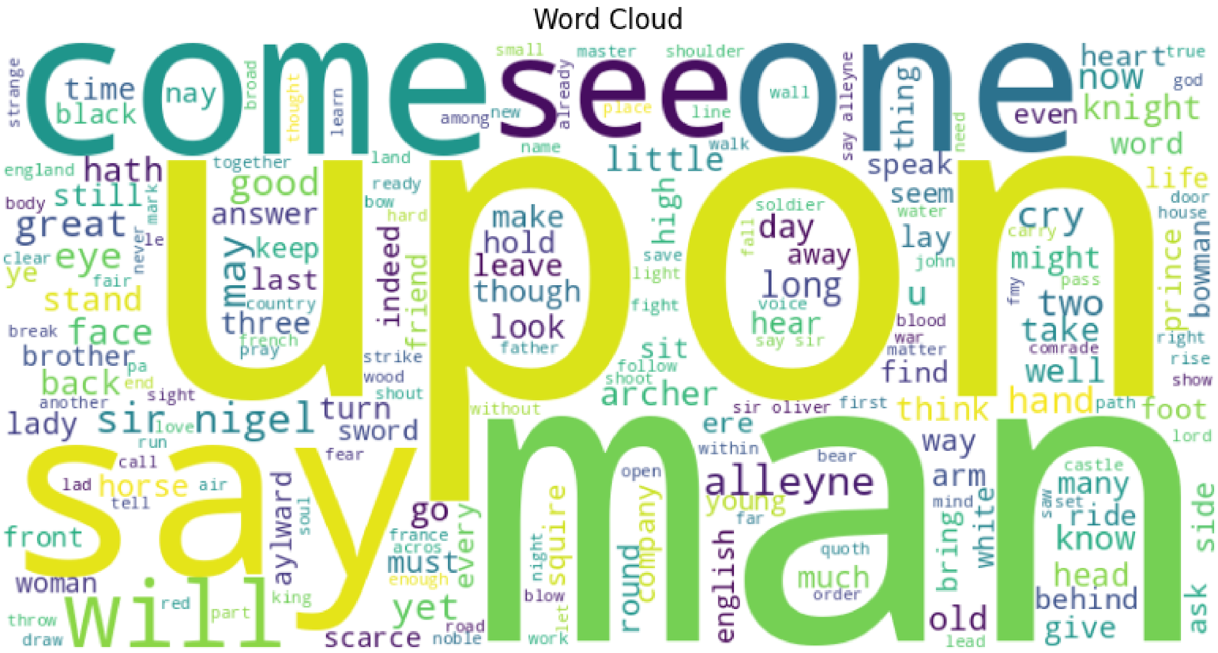
```python
# Plotting tag frequency (Treebank Tags)
to_plot = []
to_plot_tag = []
for sentence in words:
    for word, tag in sentence:
        to_plot_tag.append(tag)

counted_tag = Counter(to_plot_tag)
tag_freq = pd.DataFrame(counted_tag.items(), columns=['tag', 'frequency']).sort_values(by='frequency', ascending=False)

tag_freq = tag_freq.head(10)
plt.title("Tag Frequency (Treebank Tags)")
sns.barplot(x='frequency', y='tag', data=tag_freq)
plt.show()
```

Displaying frequencies:

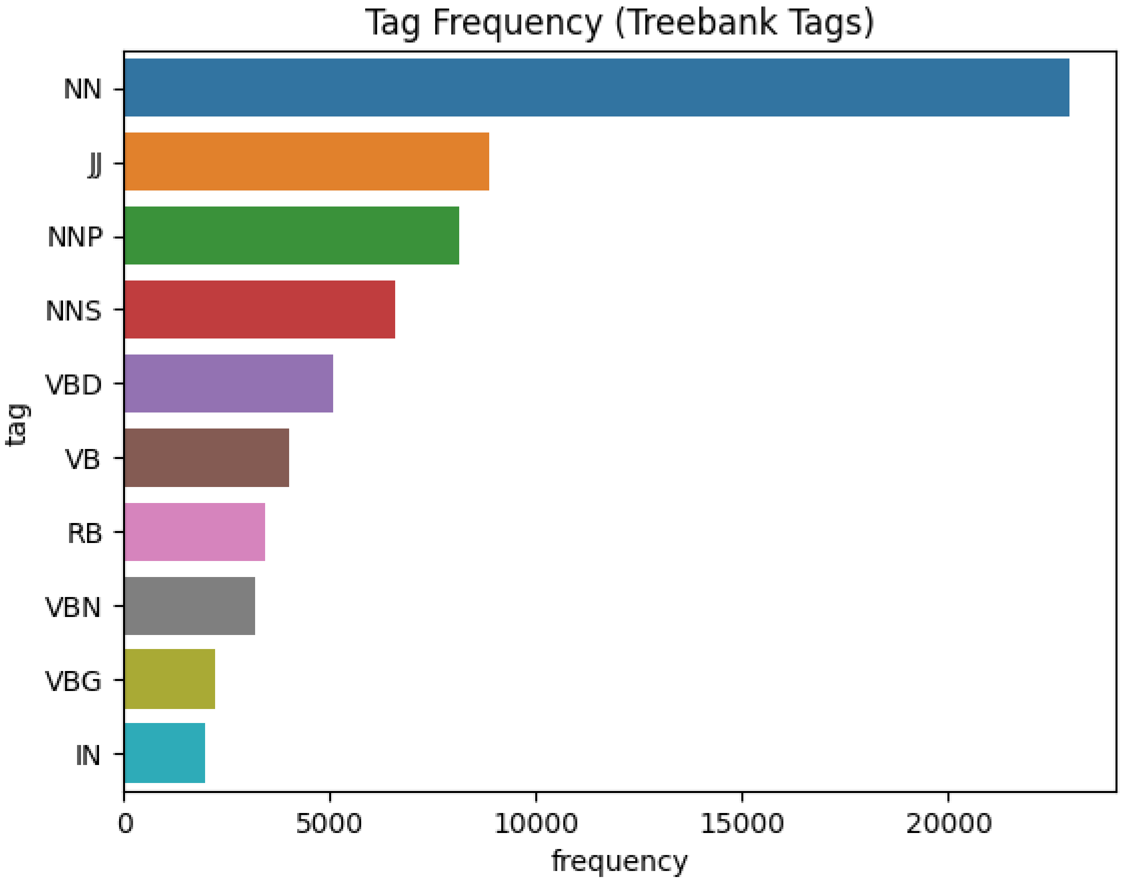
- Displaying the word frequency:



- **Word Cloud:**


Word Cloud
come
see
one
upon
man
say
will
time
black
nay
heart
true
now
knight
word
small
master
shoulder
god
strange
broad
thought
learn
already
alleyne
thing
even
among
new
place
line
wall
say alleyne
need
england
hath
together
land
name
walk
little
speak
life
good
still
ready
bow
soldier
seem
door
house
body
great
answer
make
water
hard
cry
carry
mark
never
le
fall
day
lay
bowman
eye
keep
hold
away
might
john
clear
ye
may
indeed
leave
save
pass
fair
last
light
long
u
prince
stand
country
though
high
fmy
two
fight
voice
face
friend
look
blood
take
break
three
hear
right
french
war
pray
father
say sir
matter
comrade
rise
brother
strike
sit
follow
pa
wood
shoot
find
well
show
back
end
turn
shout
archer
hand
another
sight
sir
nigel
without
sir oliver
first
think
path
foot
lady
sword
ere
way
lord
love
run
fear
within
bear
call
arm
castle
horse
open
alleyne
many
lad
air
soul
mind
set
tell
saw
young
ride
go
far
front
aylward
every
company
white
know
france
across
night
squire
english
quoth
must
bring
side
enough
round
head
woman
much
king
blow
order
behind
red
part
yet
let
old
ask
throw
road
give
draw
scarce
noble
work
lead

- **Tag frequency:**



Finding the Largest Chapter in the Book:

- The code begins by importing necessary libraries: 're' for regular expressions and 'defaultdict' for a dictionary with default values.

- It opens and reads the content of a text file located at '/book1.txt' into the 'data' variable.

- Splitting Chapters:
  - The code assumes that chapters in the book are separated by the text "Chapter {number}".

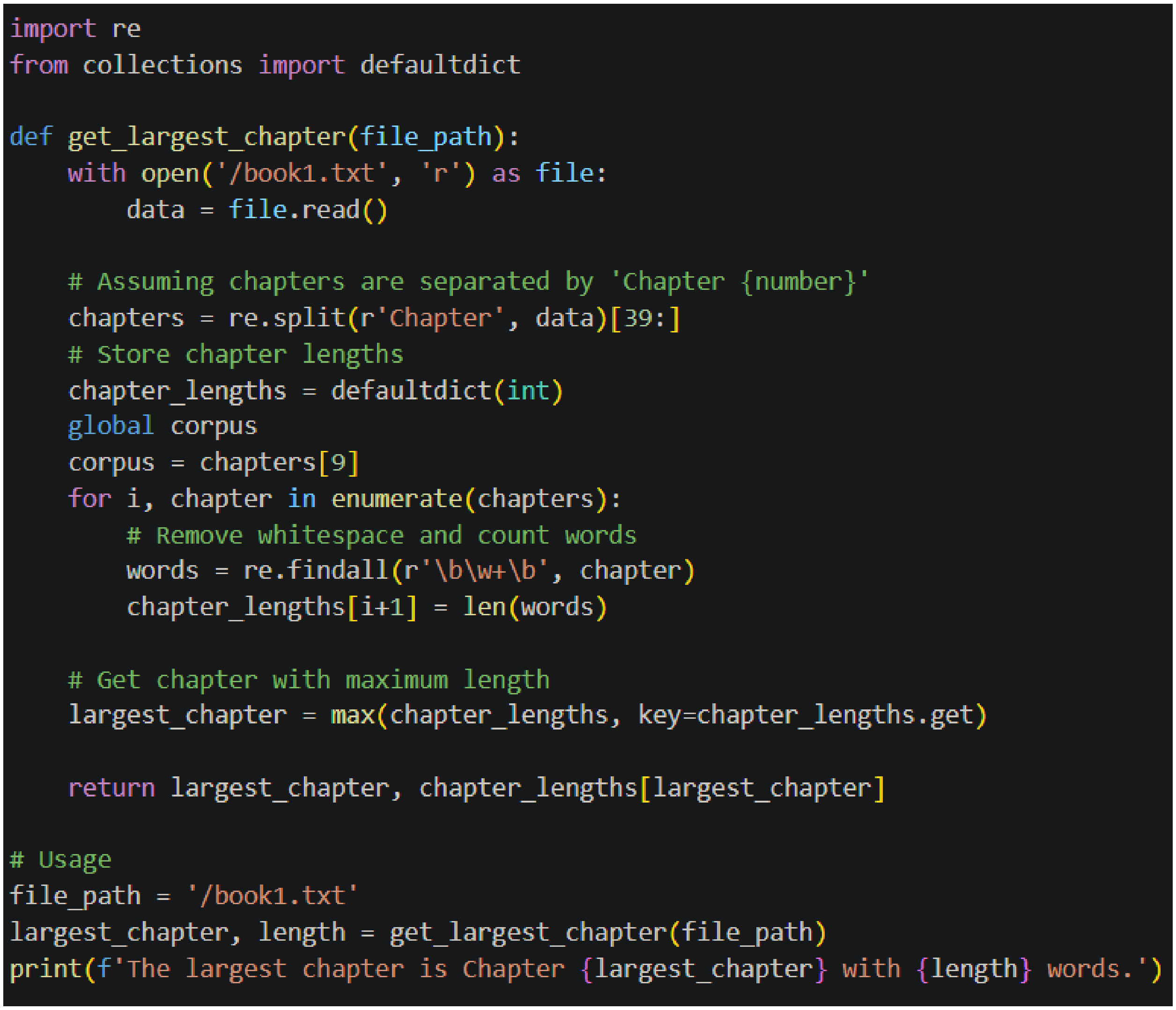
    - It uses a regular expression ('re.split()') to split the text into chapters. The '[39:]' slice is used to exclude content before the 40th occurrence of "Chapter," which is used to skip any table of contents or preambles.

- Storing Chapter Lengths:
    - It initializes a dictionary called 'chapter_lengths' to store the length (word count) of each chapter.
    - A global variable 'corpus' is initialized with the 10th chapter's content.

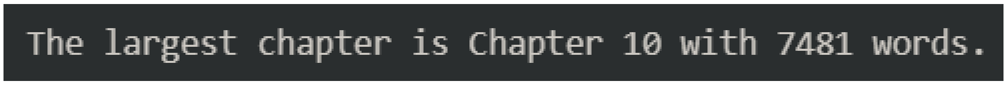
- The code iterates through the chapters, counting the words in each chapter and storing the counts in the 'chapter_lengths' dictionary.

- It identifies the chapter with the maximum word count using the 'max()' function and stores the chapter number and word count in 'largest_chapter' and 'length', respectively.

- The function returns the chapter number and word count of the largest chapter.



```python
import re
from collections import defaultdict

def get_largest_chapter(file_path):
    with open('/book1.txt', 'r') as file:
        data = file.read()

    # Assuming chapters are separated by 'Chapter {number}'
    chapters = re.split(r'Chapter', data)[39:]
    # Store chapter lengths
    chapter_lengths = defaultdict(int)
    global corpus
    corpus = chapters[9]
    for i, chapter in enumerate(chapters):
        # Remove whitespace and count words
        words = re.findall(r'\b\w+\b', chapter)
        chapter_lengths[i+1] = len(words)

    # Get chapter with maximum length
    largest_chapter = max(chapter_lengths, key=chapter_lengths.get)

    return largest_chapter, chapter_lengths[largest_chapter]

# Usage
file_path = '/book1.txt'
largest_chapter, length = get_largest_chapter(file_path)
print(f'The largest chapter is Chapter {largest_chapter} with {length} words.')
```



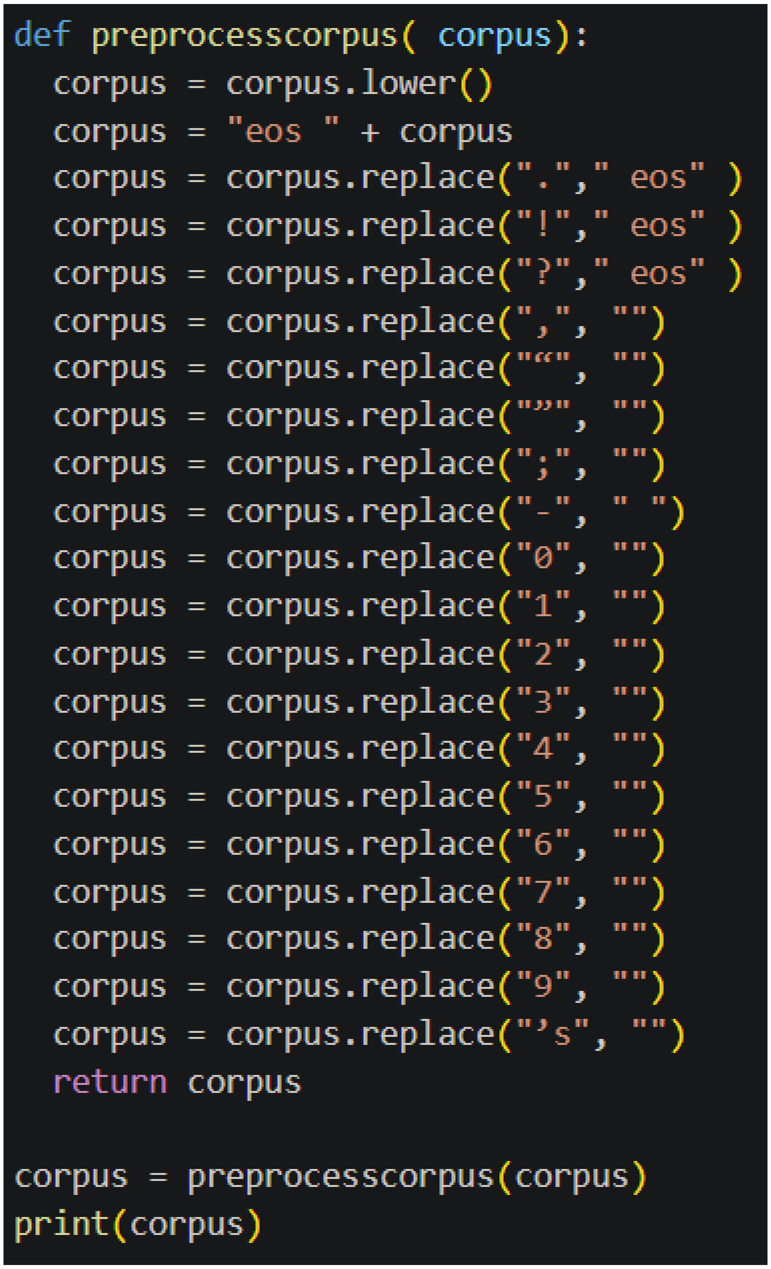
```
The largest chapter is Chapter 10 with 7481 words.
```

Text Preprocessing Function:

This function, 'preprocesscorpus(corpus)', performs several text preprocessing steps on an input text corpus:

- Converts text to lowercase.
- Adds "eos" as a sentence boundary marker at the beginning.
- Replace periods, exclamation marks, and question marks with "eos" to mark the end of sentences.
- Removes commas, quotation marks, semicolons, hyphens, and digits.

- Removes the English possessive form "'s".



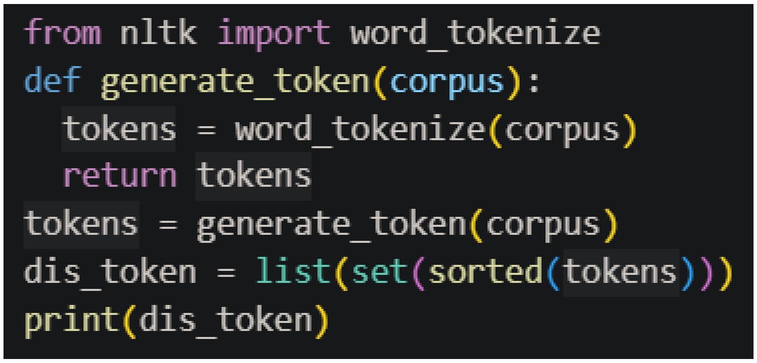
```
def preprocesscorpus( corpus):
  corpus = corpus.lower()
  corpus = "eos " + corpus
  corpus = corpus.replace("."," eos" )
  corpus = corpus.replace("!"," eos" )
  corpus = corpus.replace("?"," eos" )
  corpus = corpus.replace(",", "")
  corpus = corpus.replace("“", "")
  corpus = corpus.replace("”", "")
  corpus = corpus.replace(";", "")
  corpus = corpus.replace("-", " ")
  corpus = corpus.replace("0", "")
  corpus = corpus.replace("1", "")
  corpus = corpus.replace("2", "")
  corpus = corpus.replace("3", "")
  corpus = corpus.replace("4", "")
  corpus = corpus.replace("5", "")
  corpus = corpus.replace("6", "")
  corpus = corpus.replace("7", "")
  corpus = corpus.replace("8", "")
  corpus = corpus.replace("9", "")
  corpus = corpus.replace("’s", "")
  return corpus

corpus = preprocesscorpus(corpus)
print(corpus)
```

Text Tokenization and Vocabulary Generation:

1. Tokenization: It uses the NLTK library's 'word_tokenize' function to split the input 'corpus' into individual words or tokens.

2. Vocabulary Generation: It creates a unique and sorted list of tokens, forming the vocabulary of the text. Duplicate tokens are removed, and the list is sorted.

3. Finally, it prints the distinct tokens, which represent the unique words in the text corpus.

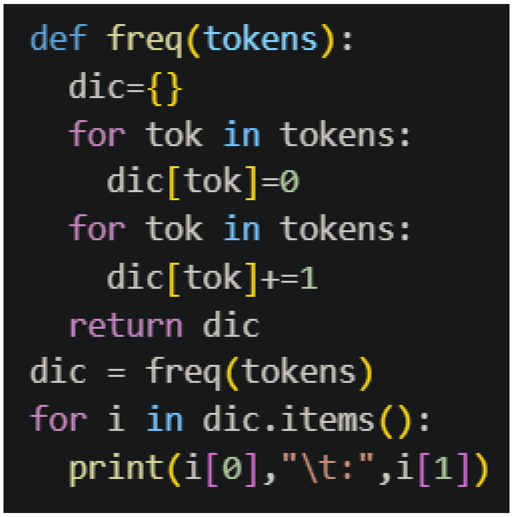


```
from nltk import word_tokenize
def generate_token(corpus):
  tokens = word_tokenize(corpus)
  return tokens
tokens = generate_token(corpus)
dis_token = list(set(sorted(tokens)))
print(dis_token)
```

Token Frequency Analysis:

- Frequency Count: It calculates the frequency of each token in the 'tokens' list and stores the results in a dictionary called 'dic'.

- Printing Token Frequencies: It then iterates through the dictionary and prints each token along with its frequency count.

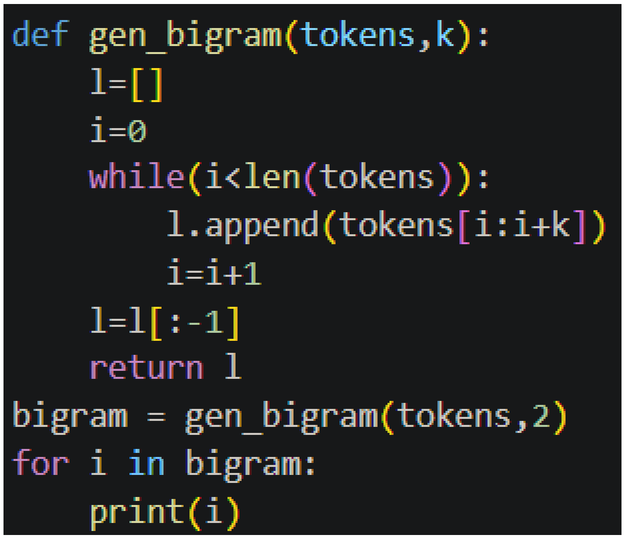


```
def freq(tokens):
  dic={}
  for tok in tokens:
    dic[tok]=0
  for tok in tokens:
    dic[tok]+=1
  return dic
dic = freq(tokens)
for i in dic.items():
  print(i[0],"\t:",i[1])
```

Generating Bigrams:

- It takes a list of tokens ('tokens') and an integer 'k' as input and generates bigrams, which are pairs of consecutive tokens. It stores these bigrams in a list 'l'.
- It iterates through the tokens, taking 'k' tokens at a time, and appends them to the 'l' list.
- It removes the last element from the 'l' list because the loop overshoots by one iteration.
- Finally, it prints the generated bigrams.

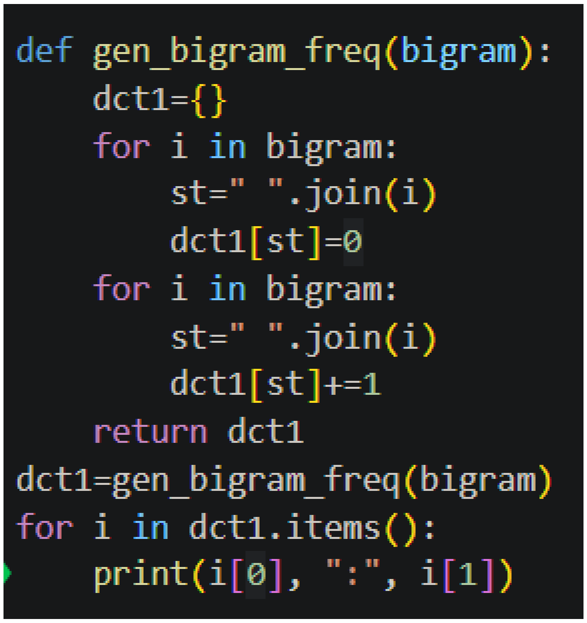
In summary, this code generates bigrams from a list of tokens and displays the resulting pairs of consecutive words.



```
def gen_bigram(tokens,k):
    l=[]
    i=0
    while(i<len(tokens)):
        l.append(tokens[i:i+k])
        i=i+1
    l=l[:-1]
    return l
bigram = gen_bigram(tokens,2)
for i in bigram:
    print(i)
```

Bigram Frequency Analysis:

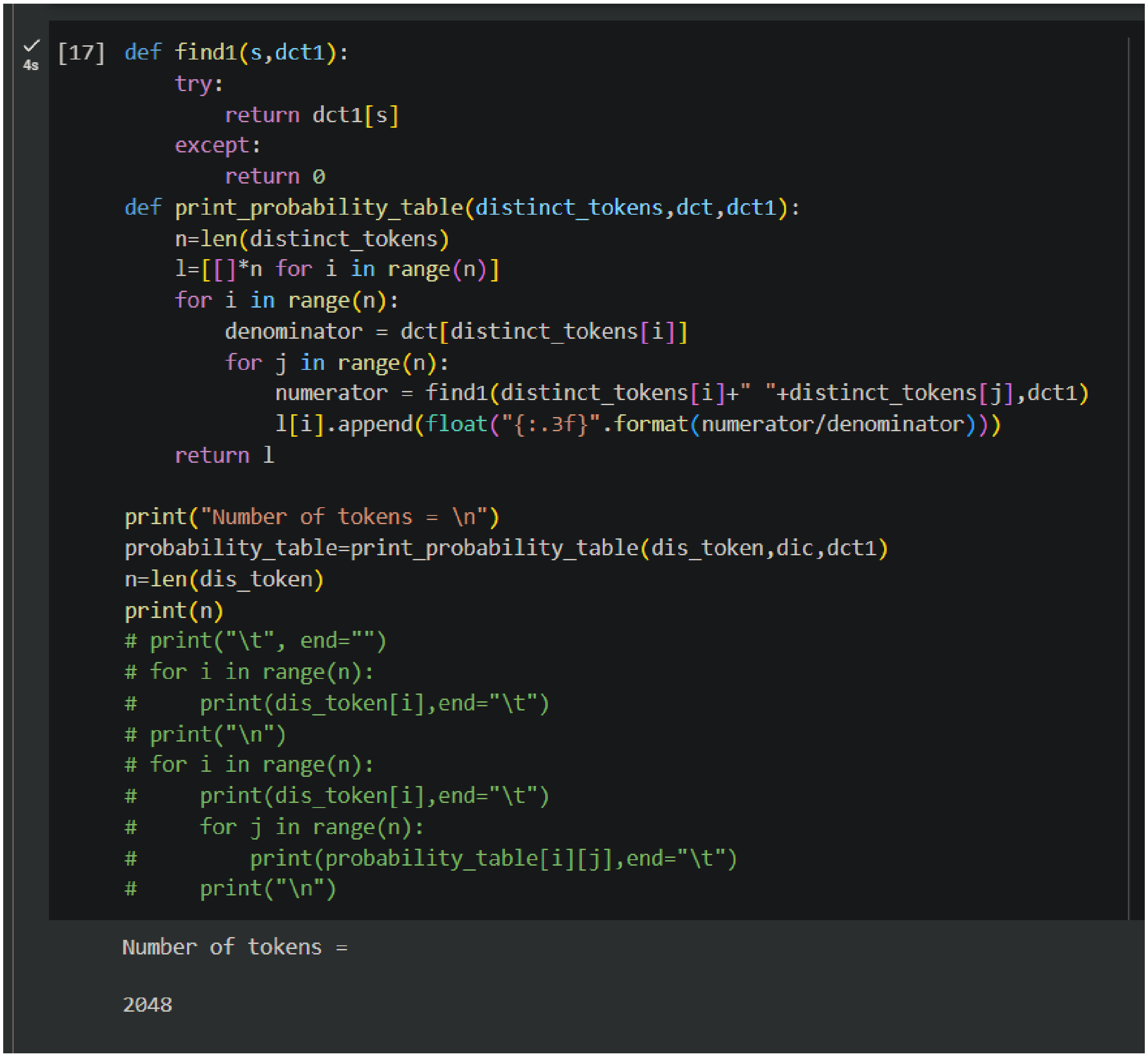
- It calculates the frequency of each unique bigram in the 'bigram' list and stores the results in a dictionary named 'dct1'.
- It then iterates through the dictionary and prints each bigram along with its frequency count.



```
def gen_bigram_freq(bigram):
    dct1={}
    for i in bigram:
        st=" ".join(i)
        dct1[st]=0
    for i in bigram:
        st=" ".join(i)
        dct1[st]+=1
    return dct1
dct1=gen_bigram_freq(bigram)
for i in dct1.items():
    print(i[0], ":", i[1])
```

Probability Table Generation:

- It calculates the conditional probabilities of word pairs based on the provided data. The resulting probabilities are stored in a two-dimensional list 'l'.
- It uses nested loops to iterate through all distinct tokens in the 'dis_token' list and calculate the conditional probabilities.
- For each pair of distinct tokens, it calculates the numerator by checking the frequency of the bigram in 'dct1' and the denominator by checking the frequency of the first token in 'dic'.
- The calculated probabilities are rounded to three decimal places.
- Finally, it prints the generated probability table.
- Note: The commented part prints probability table.



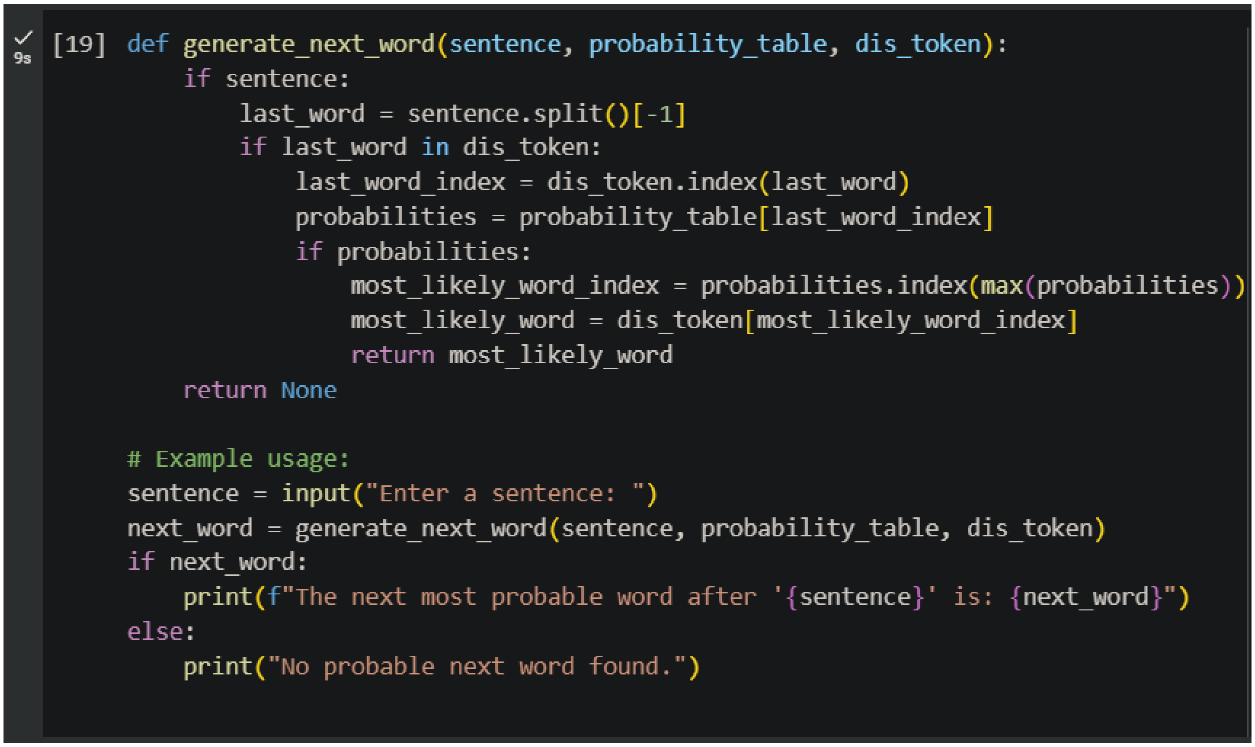
```
def find1(s,dct1):
    try:
        return dct1[s]
    except:
        return 0
def print_probability_table(distinct_tokens,dct,dct1):
    n=len(distinct_tokens)
    l=[[]*n for i in range(n)]
    for i in range(n):
        denominator = dct[distinct_tokens[i]]
        for j in range(n):
            numerator = find1(distinct_tokens[i]+" "+distinct_tokens[j],dct1)
            l[i].append(float("{:.3f}".format(numerator/denominator)))
    return l

print("Number of tokens = \n")
probability_table=print_probability_table(dis_token,dic,dct1)
n=len(dis_token)
print(n)
# print("\t", end="")
# for i in range(n):
#     print(dis_token[i],end="\t")
# print("\n")
# for i in range(n):
#     print(dis_token[i],end="\t")
#     for j in range(n):
#         print(probability_table[i][j],end="\t")
#     print("\n")
```

```
Number of tokens = 

2048
```

Bigram Word Predictor:

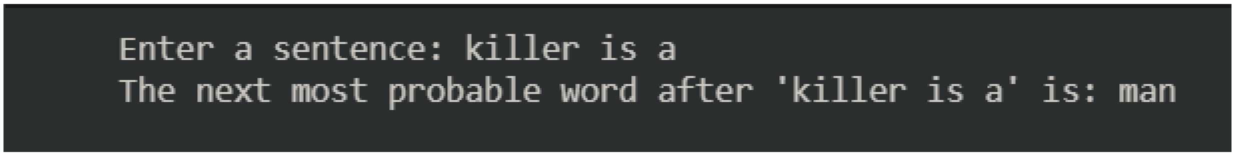
- It takes as input:
    - 'sentence': The input sentence for which you want to predict the next word.
    - 'probability_table': A table of conditional probabilities between pairs of words.
    - 'dis_token': A list of distinct tokens (words).
- It checks if there's a non-empty input sentence ('if sentence:').

- If the last word of the input sentence is in the list of distinct tokens ('dis_token'), it calculates the most likely next word based on the bigram probabilities in the 'probability_table'.
- It returns the most likely word as the prediction or 'None' if no prediction can be made.
- The code then takes user input for a sentence and uses the 'generate_next_word' function to predict the next word. If a prediction is made, it's printed; otherwise, it indicates that no probable next word was found.
- The code calculates conditional probabilities based on the bigram model. It assumes that the most probable next word depends only on the previous word (bigram), which might not be accurate for all types of text.

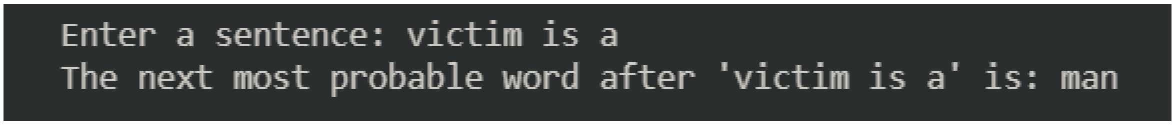


```python
def generate_next_word(sentence, probability_table, dis_token):
    if sentence:
        last_word = sentence.split()[-1]
        if last_word in dis_token:
            last_word_index = dis_token.index(last_word)
            probabilities = probability_table[last_word_index]
            if probabilities:
                most_likely_word_index = probabilities.index(max(probabilities))
                most_likely_word = dis_token[most_likely_word_index]
                return most_likely_word
    return None

# Example usage:
sentence = input("Enter a sentence: ")
next_word = generate_next_word(sentence, probability_table, dis_token)
if next_word:
    print(f"The next most probable word after '{sentence}' is: {next_word}")
else:
    print("No probable next word found.")
```

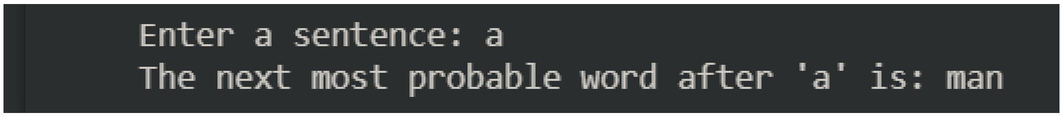
- It assumes that the most probable next word depends only on the previous word (bigram), which might not be accurate for all types of text. Hence the phrases "killer is a," "victim is a," and "a" all give the same output.



```
Enter a sentence: killer is a
The next most probable word after 'killer is a' is: man
```



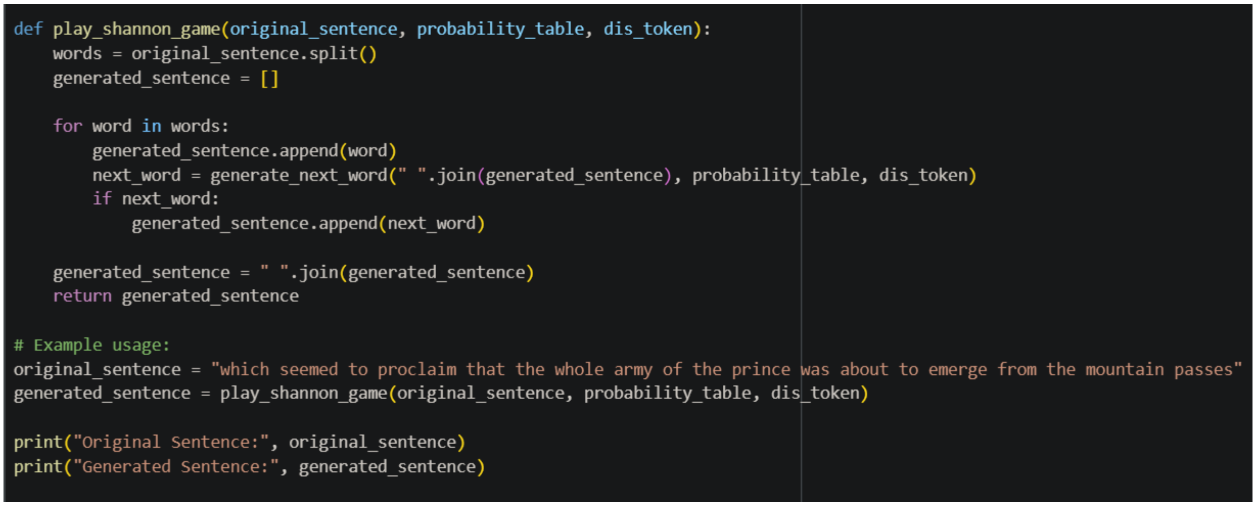
```
Enter a sentence: victim is a
The next most probable word after 'victim is a' is: man
```



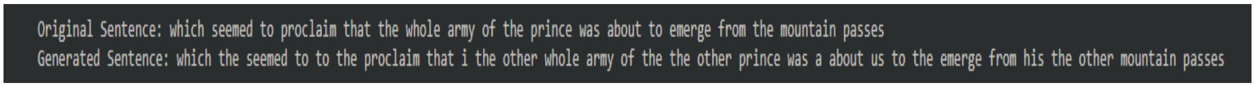
```
Enter a sentence: a
The next most probable word after 'a' is: man
```

Shannon Game Word Generation:

- The 'play_shannon_game' function takes three input parameters:
  - the 'original_sentence,' which serves as the starting point for word generation,
  - a 'probability_table' containing conditional probabilities between word pairs (based on a bigram model),
  - and 'dis_token,' a list of distinct words used for word prediction.
- It initializes an empty list called 'generated_sentence' and splits the original sentence into individual words. Then, it iterates through these words, appending each one to 'generated_sentence' and using the 'generate_next_word' function to predict the next word based on the words accumulated so far.
- After generating words for all input words, the code joins them to form a new sentence, which is returned as the function's output.



```python
def play_shannon_game(original_sentence, probability_table, dis_token):
    words = original_sentence.split()
    generated_sentence = []

    for word in words:
        generated_sentence.append(word)
        next_word = generate_next_word(" ".join(generated_sentence), probability_table, dis_token)
        if next_word:
            generated_sentence.append(next_word)

    generated_sentence = " ".join(generated_sentence)
    return generated_sentence

# Example usage:
original_sentence = "which seemed to proclaim that the whole army of the prince was about to emerge from the mountain passes"
generated_sentence = play_shannon_game(original_sentence, probability_table, dis_token)

print("Original Sentence:", original_sentence)
print("Generated Sentence:", generated_sentence)
```



```
Original Sentence: which seemed to proclaim that the whole army of the prince was about to emerge from the mountain passes
Generated Sentence: which the seemed to to the proclaim that i the other whole army of the the other prince was a about us to the emerge from his the other mountain passes
```

Bigram models are limited because they only consider the probability of a word based on the immediately preceding word. Even though two consecutive words make sense, the sentence as a whole does not make sense.