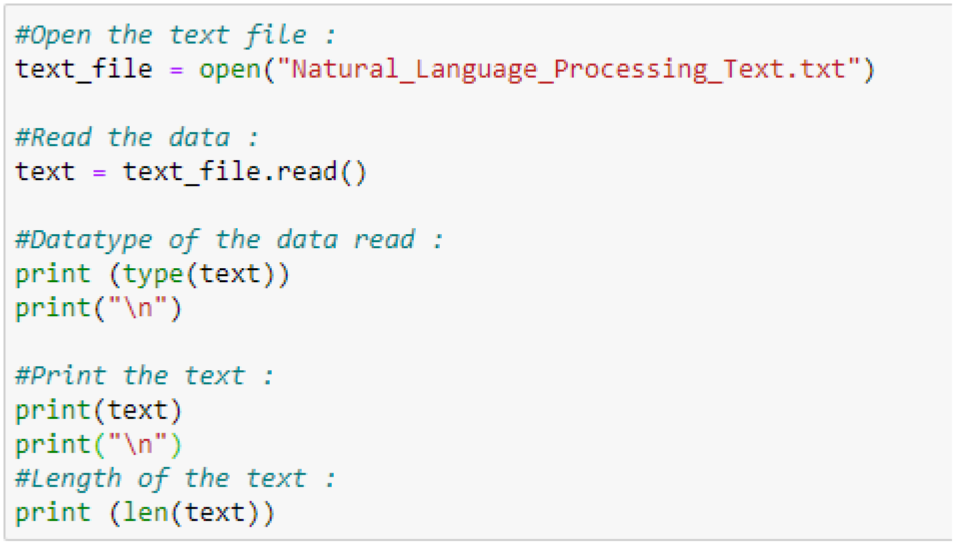
Natural Language Processing Lab

Program 1: Exploring Features of NLTK Library

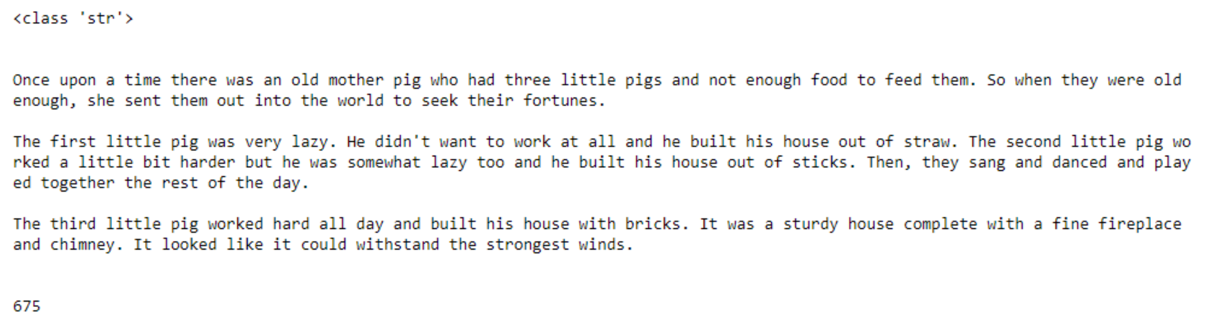
a. Open the text file for processing:

First, we are going to open and read the file which we want to analyze.



```
#Open the text file :
text_file = open("Natural_Language_Processing_Text.txt")

#Read the data :
text = text_file.read()

#Datatype of the data read :
print (type(text))
print("\n")

#Print the text :
print(text)
print("\n")
#Length of the text :
print (len(text))
```

Figure 1: Small code snippet to open and read the text file and analyze it.



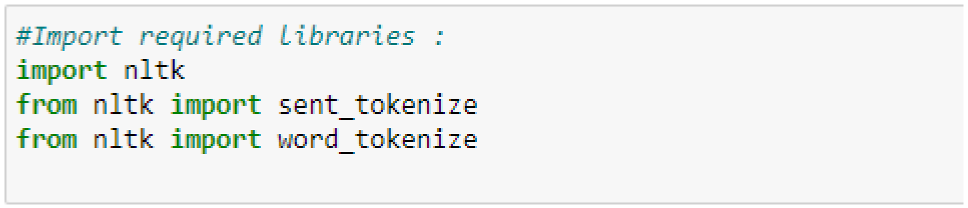
```
<class 'str'>


Once upon a time there was an old mother pig who had three little pigs and not enough food to feed them. So when they were old
enough, she sent them out into the world to seek their fortunes.

The first little pig was very lazy. He didn't want to work at all and he built his house out of straw. The second little pig wo
rked a little bit harder but he was somewhat lazy too and he built his house out of sticks. Then, they sang and danced and play
ed together the rest of the day.

The third little pig worked hard all day and built his house with bricks. It was a sturdy house complete with a fine fireplace
and chimney. It looked like it could withstand the strongest winds.


675
```

Figure 2: Text string file.

Next, notice that the data type of the text file read is a **String**. The number of characters in our text file is **675**.

b. Import required libraries:

For various data processing cases in NLP, we need to import some libraries. In this case, we are going to use NLTK for Natural Language Processing. We will use it to perform various operations on the text.

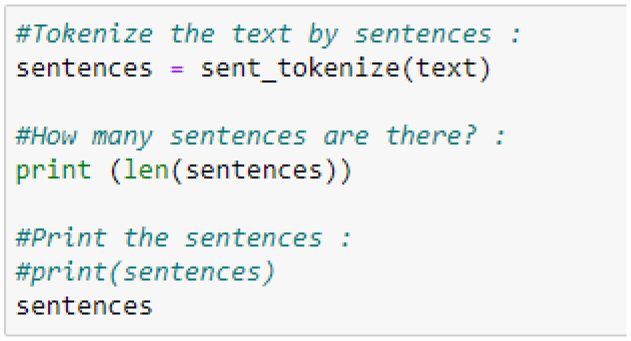


```python
#Import required libraries :
import nltk
from nltk import sent_tokenize
from nltk import word_tokenize
```

Figure 3: Importing the required libraries.

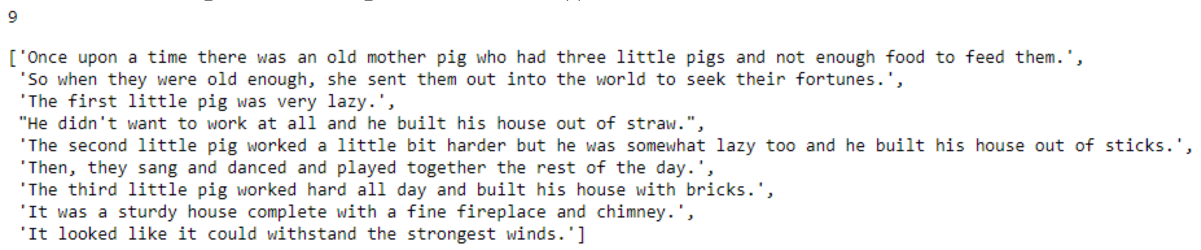
c. Sentence tokenizing:

By tokenizing the text with sent_tokenize( ), we can get the text as sentences.



```python
#Tokenize the text by sentences :
sentences = sent_tokenize(text)

#How many sentences are there? :
print (len(sentences))

#Print the sentences :
#print(sentences)
sentences
```

Figure 4: Using sent_tokenize( ) to tokenize the text as sentences.



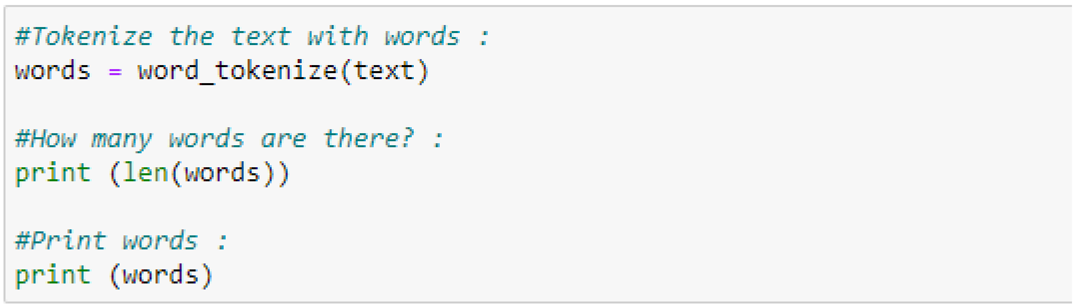
```
9

['Once upon a time there was an old mother pig who had three little pigs and not enough food to feed them.',
 'So when they were old enough, she sent them out into the world to seek their fortunes.',
 'The first little pig was very lazy.',
 "He didn't want to work at all and he built his house out of straw.",
 'The second little pig worked a little bit harder but he was somewhat lazy too and he built his house out of sticks.',
 'Then, they sang and danced and played together the rest of the day.',
 'The third little pig worked hard all day and built his house with bricks.',
 'It was a sturdy house complete with a fine fireplace and chimney.',
 'It looked like it could withstand the strongest winds.']
```

Figure 5: Text sample data.

In the example above, we can see the entire text of our data is represented as sentences and also notice that the total number of sentences here is **9**.

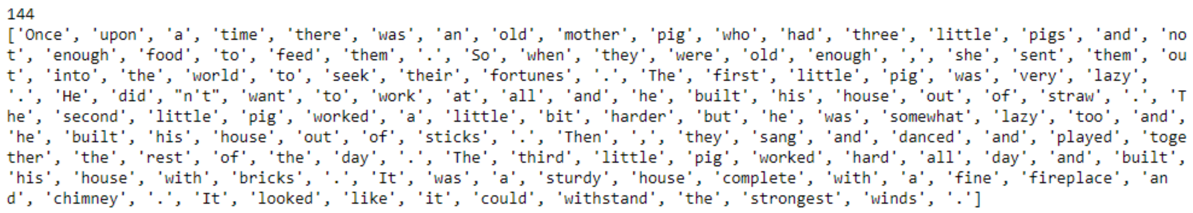
d. Word tokenizing:

By tokenizing the text with word_tokenize( ), we can get the text as words.



```python
#Tokenize the text with words :
words = word_tokenize(text)

#How many words are there? :
print (len(words))

#Print words :
print (words)
```

Figure 6: Using word_tokenize() to tokenize the text as words.



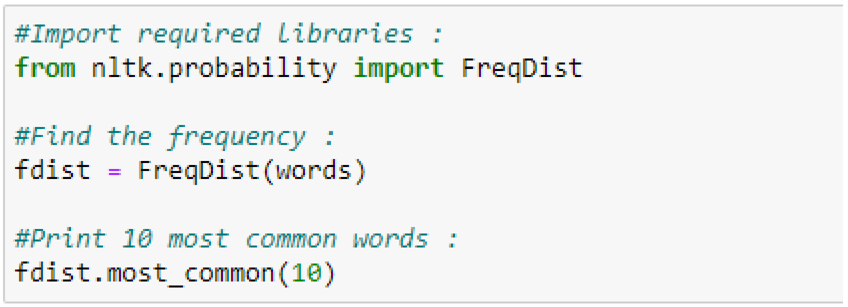
```
144
['Once', 'upon', 'a', 'time', 'there', 'was', 'an', 'old', 'mother', 'pig', 'who', 'had', 'three', 'little', 'pigs', 'and', 'no
t', 'enough', 'food', 'to', 'feed', 'them', '.', 'So', 'when', 'they', 'were', 'old', 'enough', ',', 'she', 'sent', 'them', 'ou
t', 'into', 'the', 'world', 'to', 'seek', 'their', 'fortunes', '.', 'The', 'first', 'little', 'pig', 'was', 'very', 'lazy',
'.', 'He', 'did', "n't", 'want', 'to', 'work', 'at', 'all', 'and', 'he', 'built', 'his', 'house', 'out', 'of', 'straw', '.', 'T
he', 'second', 'little', 'pig', 'worked', 'a', 'little', 'bit', 'harder', 'but', 'he', 'was', 'somewhat', 'lazy', 'too', 'and',
'he', 'built', 'his', 'house', 'out', 'of', 'sticks', '.', 'Then', ',', 'they', 'sang', 'and', 'danced', 'and', 'played', 'toge
ther', 'the', 'rest', 'of', 'the', 'day', '.', 'The', 'third', 'little', 'pig', 'worked', 'hard', 'all', 'day', 'and', 'built',
'his', 'house', 'with', 'bricks', '.', 'It', 'was', 'a', 'sturdy', 'house', 'complete', 'with', 'a', 'fine', 'fireplace', 'an
d', 'chimney', '.', 'It', 'looked', 'like', 'it', 'could', 'withstand', 'the', 'strongest', 'winds', '.']
```

Figure 7: Text sample data.

Next, we can see the entire text of our data is represented as words and also notice that the total number of words here is **144**.

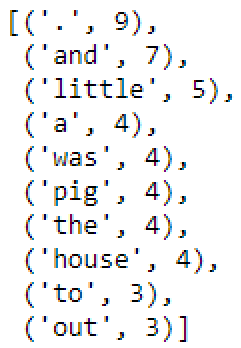
e. Find the frequency distribution:

Let's find out the frequency of words in our text.



```python
#Import required libraries :
from nltk.probability import FreqDist

#Find the frequency :
fdist = FreqDist(words)

#Print 10 most common words :
fdist.most_common(10)
```

Figure 8: Using FreqDist() to find the frequency of words in our sample text.



```
[('.', 9),
 ('and', 7),
 ('little', 5),
 ('a', 4),
 ('was', 4),
 ('pig', 4),
 ('the', 4),
 ('house', 4),
 ('to', 3),
 ('out', 3)]
```

Figure 9: Printing the ten most common words from the sample text.

Notice that the most used words are punctuation marks and stopwords. We will have to remove such words to analyze the actual text.

f. Plot the frequency graph:

Let's plot a graph to visualize the word distribution in our text.

```
#Plot the graph for fdist :
import matplotlib.pyplot as plt

fdist.plot(10)
```

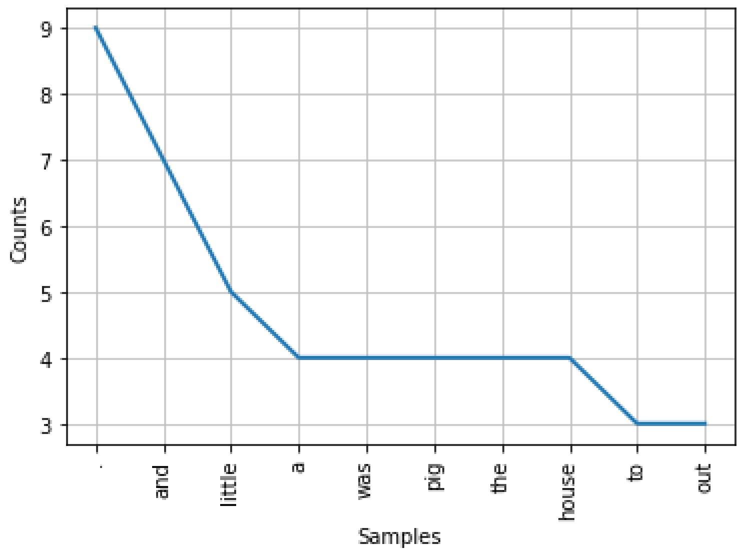
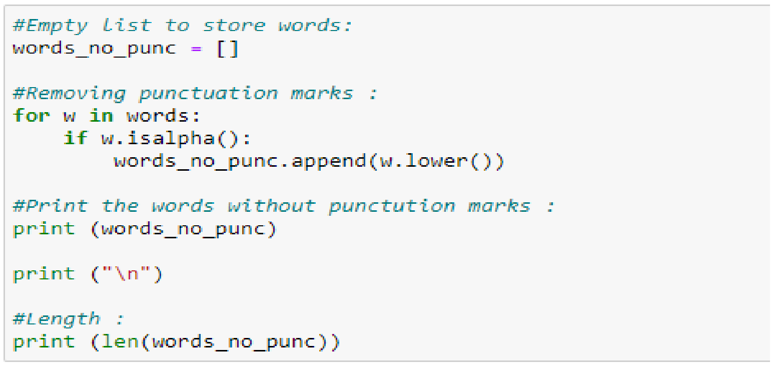


Figure 10: Plotting a graph to visualize the text distribution.

In the graph above, notice that a period "." is used nine times in our text. Analytically speaking, punctuation marks are not that important for natural language processing. Therefore, in the next step, we will be removing such punctuation marks.

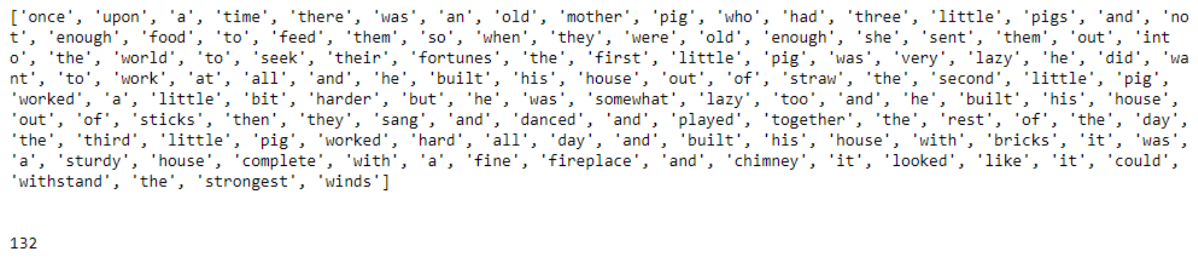
g. Remove punctuation marks:

Next, we are going to remove the punctuation marks as they are not very useful for us. We are going to use isalpha( ) method to separate the punctuation marks from the actual text. Also, we are going to make a new list called words_no_punc, which will store the words in lower case but exclude the punctuation marks.



```
#Empty List to store words:
words_no_punc = []

#Removing punctuation marks :
for w in words:
    if w.isalpha():
        words_no_punc.append(w.lower())

#Print the words without punctution marks :
print (words_no_punc)

print ("\n")

#Length :
print (len(words_no_punc))
```

Figure 11: Using the isalpha() method to separate the punctuation marks, along with creating a list under words_no_punc to separate words with no punctuation marks.

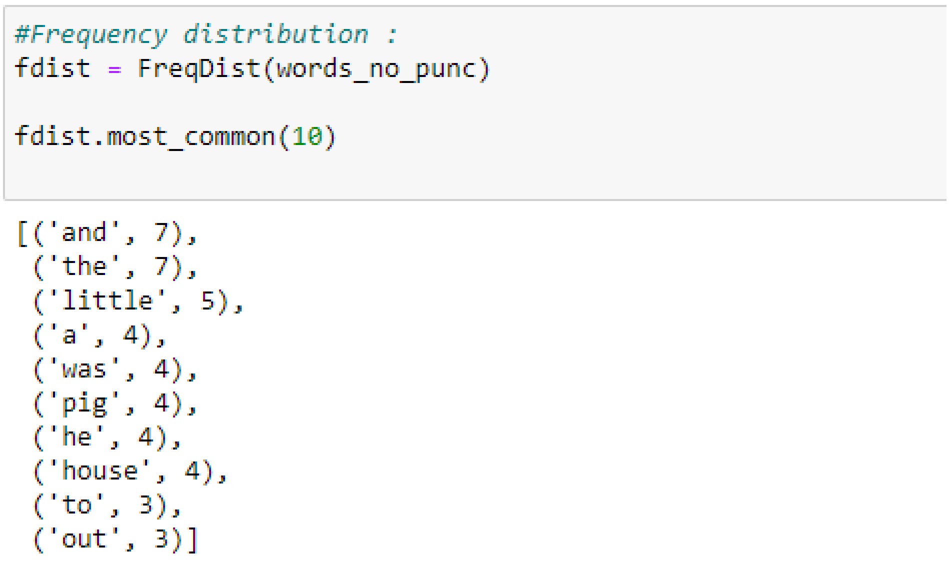


```
['once', 'upon', 'a', 'time', 'there', 'was', 'an', 'old', 'mother', 'pig', 'who', 'had', 'three', 'little', 'pigs', 'and', 'no
t', 'enough', 'food', 'to', 'feed', 'them', 'so', 'when', 'they', 'were', 'old', 'enough', 'she', 'sent', 'them', 'out', 'int
o', 'the', 'world', 'to', 'seek', 'their', 'fortunes', 'the', 'first', 'little', 'pig', 'was', 'very', 'lazy', 'he', 'did', 'wa
nt', 'to', 'work', 'at', 'all', 'and', 'he', 'built', 'his', 'house', 'out', 'of', 'straw', 'the', 'second', 'little', 'pig',
'worked', 'a', 'little', 'bit', 'harder', 'but', 'he', 'was', 'somewhat', 'lazy', 'too', 'and', 'he', 'built', 'his', 'house',
'out', 'of', 'sticks', 'then', 'they', 'sang', 'and', 'danced', 'and', 'played', 'together', 'the', 'rest', 'of', 'the', 'day',
'the', 'third', 'little', 'pig', 'worked', 'hard', 'all', 'day', 'and', 'built', 'his', 'house', 'with', 'bricks', 'it', 'was',
'a', 'sturdy', 'house', 'complete', 'with', 'a', 'fine', 'fireplace', 'and', 'chimney', 'it', 'looked', 'like', 'it', 'could',
'withstand', 'the', 'strongest', 'winds']


132
```

Figure 12: Text sample data.

As shown above, all the punctuation marks from our text are excluded. These can also cross-check with the number of words.

h. Plotting graph without punctuation marks:



```
#Frequency distribution :
fdist = FreqDist(words_no_punc)

fdist.most_common(10)
```

```
[('and', 7),
 ('the', 7),
 ('little', 5),
 ('a', 4),
 ('was', 4),
 ('pig', 4),
 ('he', 4),
 ('house', 4),
 ('to', 3),
 ('out', 3)]
```

Figure 13: Printing the ten most common words from the sample text.

```
#Plot the most common words on grpah:

fdist.plot(10)
```

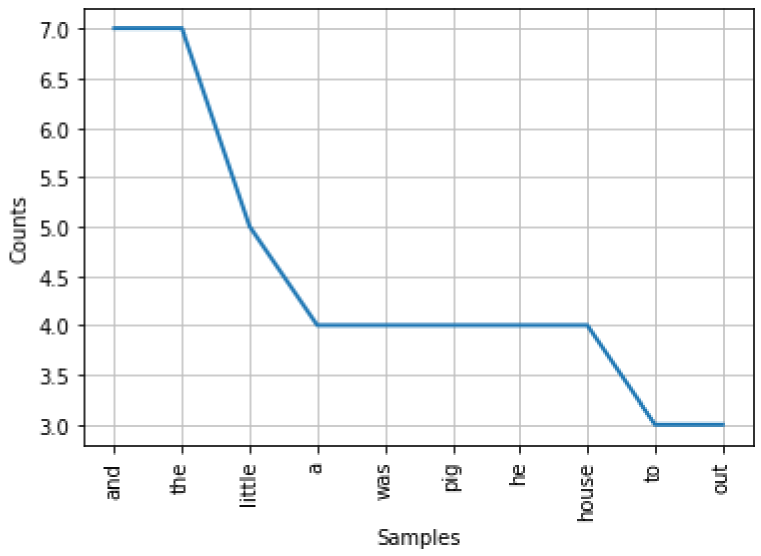


Figure 14: Plotting the graph without punctuation marks.

Notice that we still have many words that are not very useful in the analysis of our text file sample, such as “and,” “but,” “so,” and others. Next, we need to remove coordinating conjunctions.

i. List of stopwords:

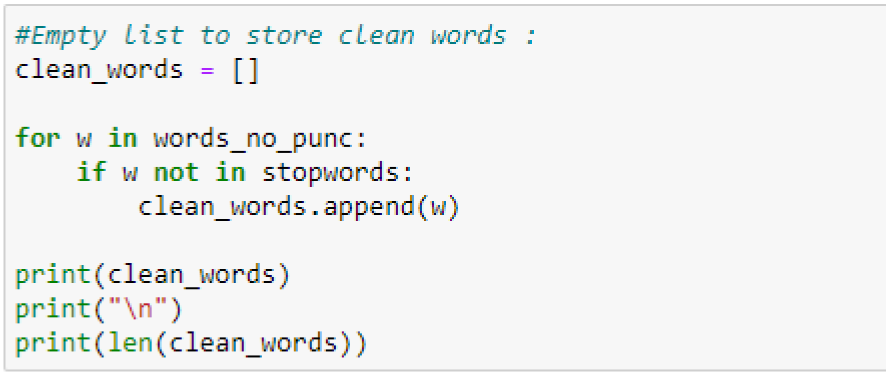
```
from nltk.corpus import stopwords

#List of stopwords
stopwords = stopwords.words("english")
print(stopwords)
```

Figure 15: Importing the list of stopwords.

```
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're", "you've", "you'll", "you'd", 'your', 'yours', 'y
ourself', 'yourselves', 'he', 'him', 'his', 'himself', 'she', "she's", 'her', 'hers', 'herself', 'it', "it's", 'its', 'itself',
'they', 'them', 'their', 'theirs', 'themselves', 'what', 'which', 'who', 'whom', 'this', 'that', "that'll", 'these', 'those',
'am', 'is', 'are', 'was', 'were', 'be', 'been', 'being', 'have', 'has', 'had', 'having', 'do', 'does', 'did', 'doing', 'a', 'a
n', 'the', 'and', 'but', 'if', 'or', 'because', 'as', 'until', 'while', 'of', 'at', 'by', 'for', 'with', 'about', 'against', 'b
etween', 'into', 'through', 'during', 'before', 'after', 'above', 'below', 'to', 'from', 'up', 'down', 'in', 'out', 'on', 'of
f', 'over', 'under', 'again', 'further', 'then', 'once', 'here', 'there', 'when', 'where', 'why', 'how', 'all', 'any', 'both',
'each', 'few', 'more', 'most', 'other', 'some', 'such', 'no', 'nor', 'not', 'only', 'own', 'same', 'so', 'than', 'too', 'very',
's', 't', 'can', 'will', 'just', 'don', "don't", 'should', "should've", 'now', 'd', 'll', 'm', 'o', 're', 've', 'y', 'ain', 'ar
en', "aren't", 'couldn', "couldn't", 'didn', "didn't", 'doesn', "doesn't", 'hadn', "hadn't", 'hasn', "hasn't", 'haven', "have
n't", 'isn', "isn't", 'ma', 'mightn', "mightn't", 'mustn', "mustn't", 'needn', "needn't", 'shan', "shan't", 'shouldn', "should
n't", 'wasn', "wasn't", 'weren', "weren't", 'won', "won't", 'wouldn', "wouldn't"]
```

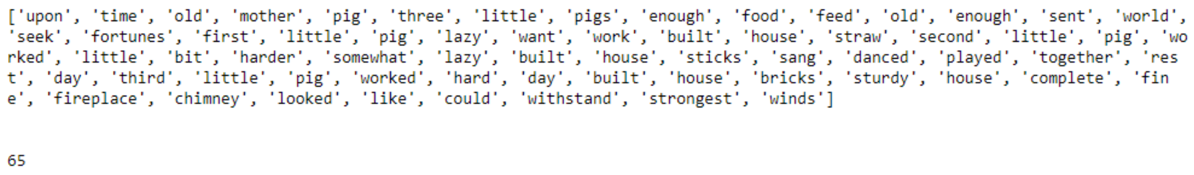
Figure 16: Text sample data.

j. Removing stopwords:



```python
#Empty list to store clean words :
clean_words = []

for w in words_no_punc:
    if w not in stopwords:
        clean_words.append(w)

print(clean_words)
print("\n")
print(len(clean_words))
```

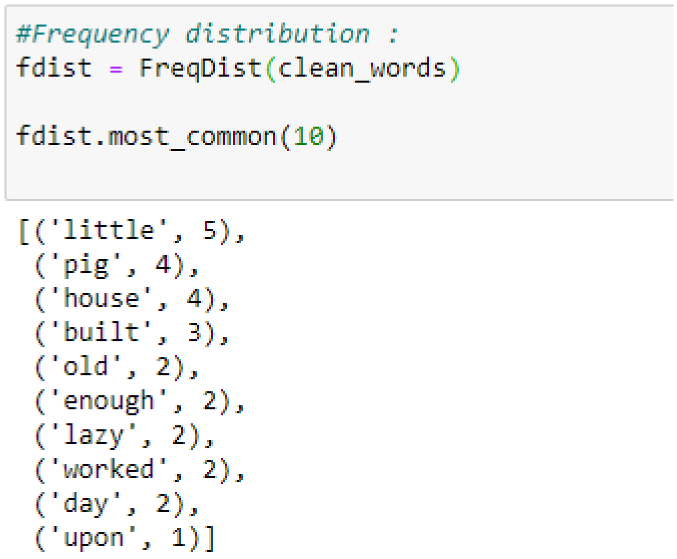
Figure 17: Cleaning the text sample data.



```
['upon', 'time', 'old', 'mother', 'pig', 'three', 'little', 'pigs', 'enough', 'food', 'feed', 'old', 'enough', 'sent', 'world',
'seek', 'fortunes', 'first', 'little', 'pig', 'lazy', 'want', 'work', 'built', 'house', 'straw', 'second', 'little', 'pig', 'wo
rked', 'little', 'bit', 'harder', 'somewhat', 'lazy', 'built', 'house', 'sticks', 'sang', 'danced', 'played', 'together', 'res
t', 'day', 'third', 'little', 'pig', 'worked', 'hard', 'day', 'built', 'house', 'bricks', 'sturdy', 'house', 'complete', 'fin
e', 'fireplace', 'chimney', 'looked', 'like', 'could', 'withstand', 'strongest', 'winds']


65
```

Figure 18: Cleaned data.

k. Final frequency distribution:



```python
#Frequency distribution :
fdist = FreqDist(clean_words)

fdist.most_common(10)
```

```
[('little', 5),
 ('pig', 4),
 ('house', 4),
 ('built', 3),
 ('old', 2),
 ('enough', 2),
 ('lazy', 2),
 ('worked', 2),
 ('day', 2),
 ('upon', 1)]
```

Figure 19: Displaying the final frequency distribution of the most common words found.

```
#Plot the most common words on grpah:

fdist.plot(10)
```

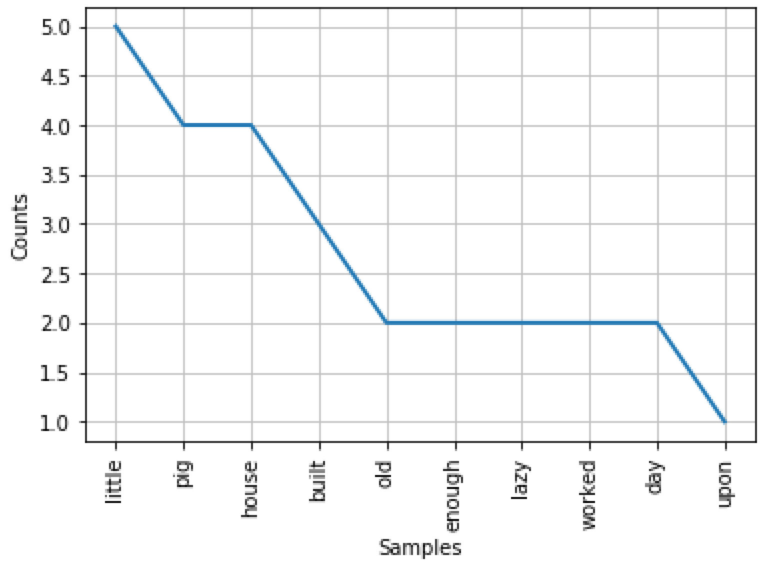


Figure 20: Visualization of the most common words found in the group.

As shown above, the final graph has many useful words that help us understand what our sample data is about, showing how essential it is to perform data cleaning on NLP.

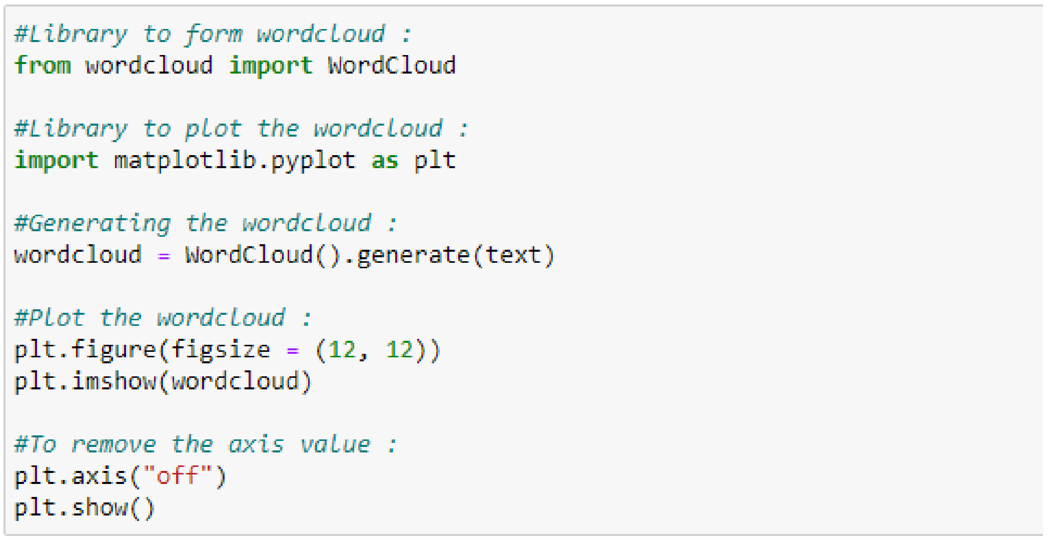
Program 2: Word Cloud

Word Cloud is a data visualization technique. In which words from a given text display on the main chart. In this technique, more frequent or essential words display in a larger and bolder font, while less frequent or essential words display in smaller or thinner fonts. It is a beneficial technique in NLP that gives us a glance at what text should be analyzed.

Properties:

1. **font_path**: It specifies the path for the fonts we want to use.
2. **width**: It specifies the width of the canvas.
3. **height**: It specifies the height of the canvas.
4. **min_font_size**: It specifies the smallest font size to use.
5. **max_font_size:** It specifies the largest font size to use.
6. **font_step**: It specifies the step size for the font.
7. **max_words**: It specifies the maximum number of words on the word cloud.
8. **stopwords**: Our program will eliminate these words.
9. **background_color:** It specifies the background color for canvas.
10. **normalize_plurals**: It removes the trailing "s" from words.

Word Cloud Python Implementation:



```
#Library to form wordcloud :
from wordcloud import WordCloud

#Library to plot the wordcloud :
import matplotlib.pyplot as plt

#Generating the wordcloud :
wordcloud = WordCloud().generate(text)

#Plot the wordcloud :
plt.figure(figsize = (12, 12))
plt.imshow(wordcloud)

#To remove the axis value :
plt.axis("off")
plt.show()
```

Figure 21: Python code implementation of the word cloud.

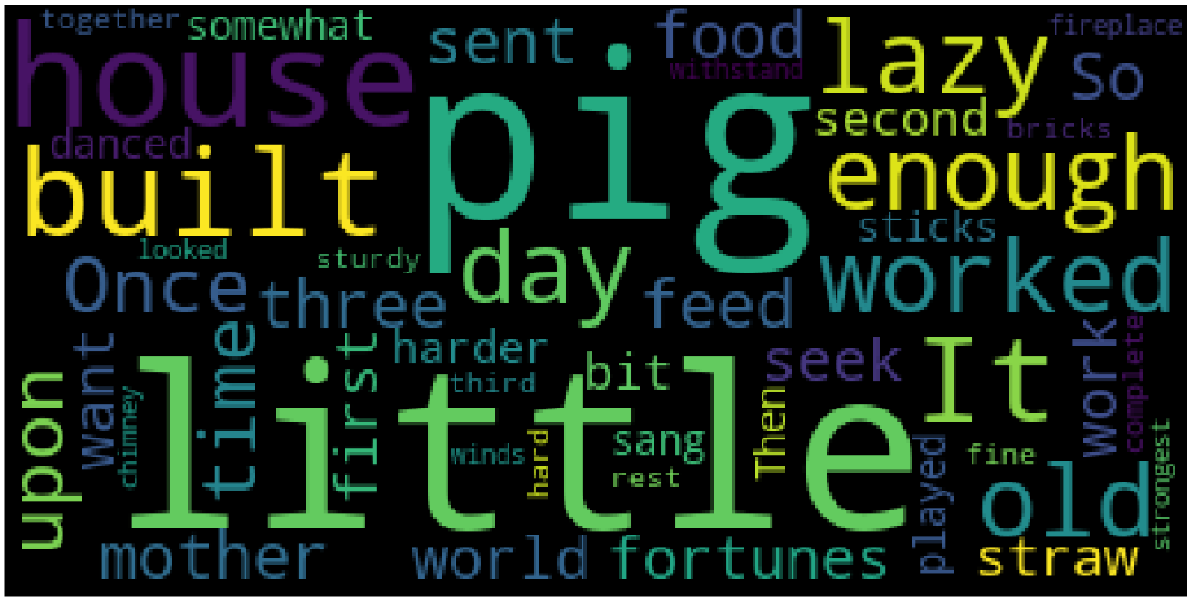


Figure 22: Word cloud example.

As shown in the graph above, the most frequent words display in larger fonts. The word cloud can be displayed in any shape or image.

For instance: In this case, we are going to use the following circle image, but we can use any shape or any image.

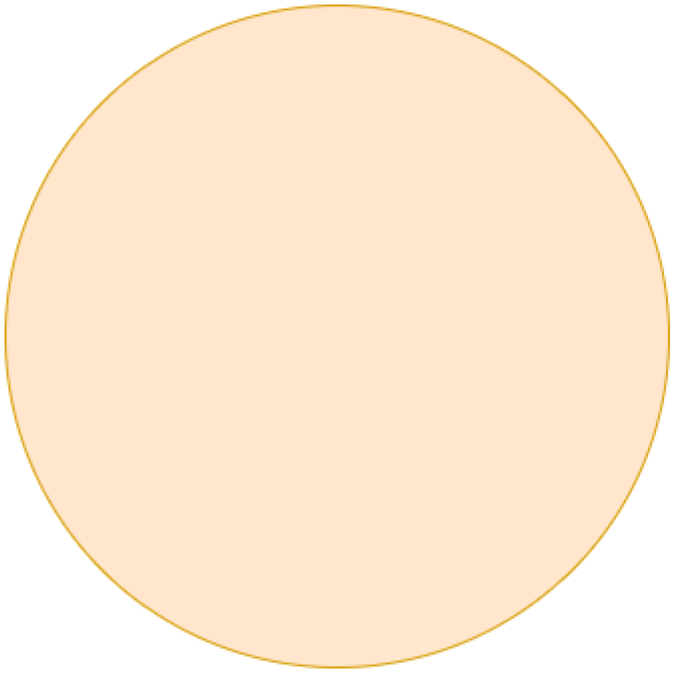
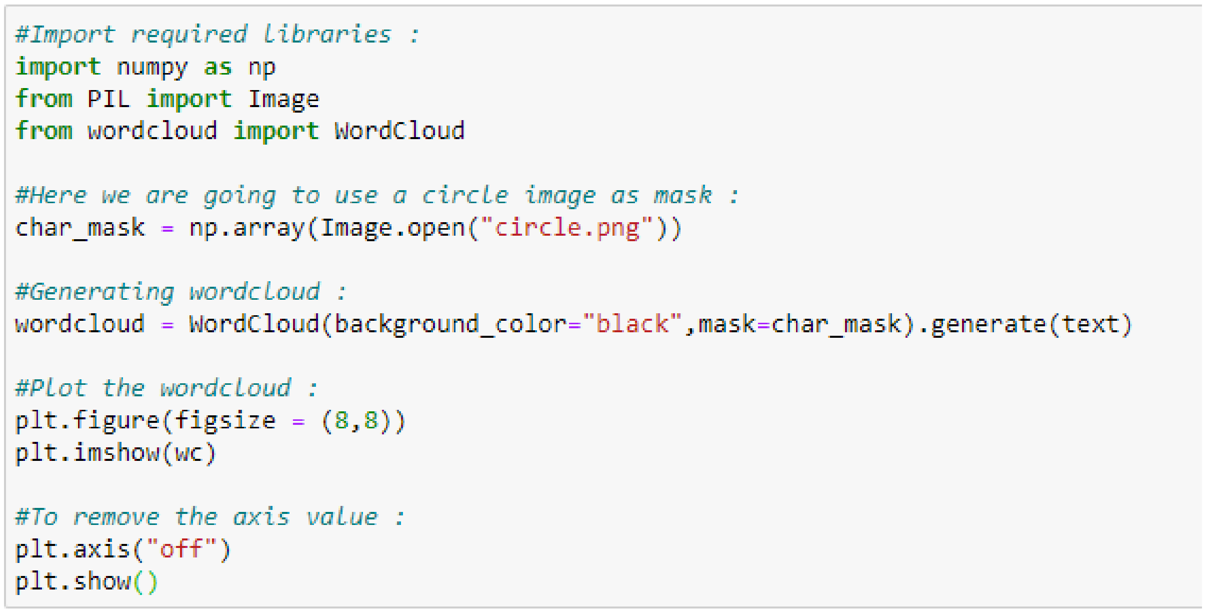


Figure 23: Circle image shape for our word cloud.
Word Cloud Python Implementation:



```
#Import required libraries :
import numpy as np
from PIL import Image
from wordcloud import WordCloud

#Here we are going to use a circle image as mask :
char_mask = np.array(Image.open("circle.png"))

#Generating wordcloud :
wordcloud = WordCloud(background_color="black",mask=char_mask).generate(text)

#Plot the wordcloud :
plt.figure(figsize = (8,8))
plt.imshow(wc)

#To remove the axis value :
plt.axis("off")
plt.show()
```

Figure 24: Python code implementation of the word cloud.

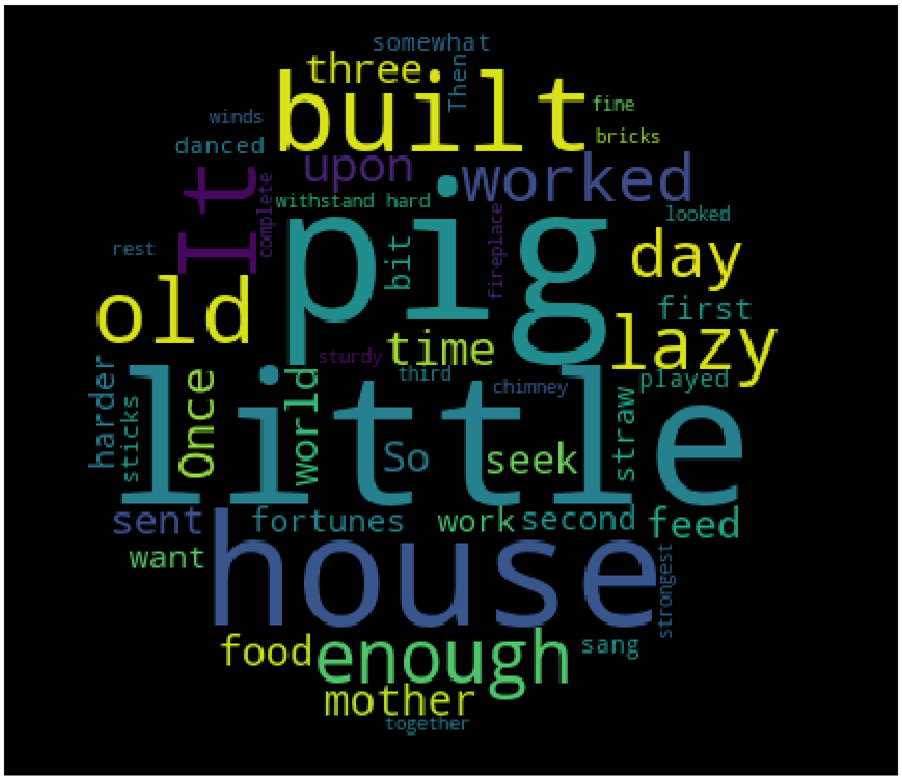


Figure 25: Word cloud with the circle shape.

As shown above, the word cloud is in the shape of a circle. As we mentioned before, we can use any shape or image to form a word cloud.

Word CloudAdvantages:

- They are fast.
- They are engaging.
- They are simple to understand.
- They are casual and visually appealing.

Word Cloud Disadvantages:

- They are non-perfect for non-clean data.
- They lack the context of words.

Program 3: Stemming

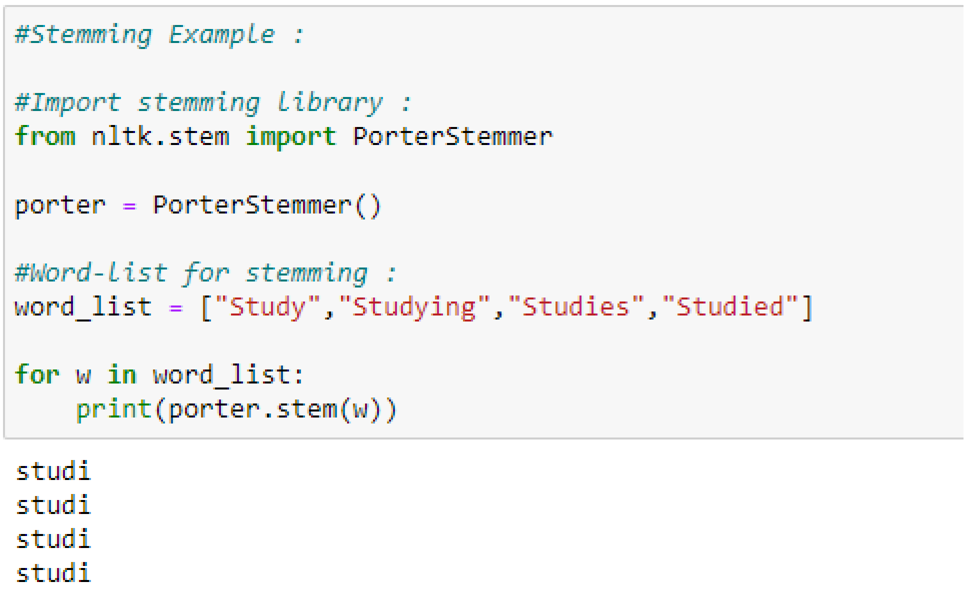
We use Stemming to normalize words. In English and many other languages, a single word can take multiple forms depending upon context used. For instance, the verb “study” can take many forms like “studies,” “studying,” “studied,” and others, depending on its context. When we tokenize words, an interpreter considers these input words as different words even though their underlying meaning is the same. Moreover, as we know that NLP is about analyzing the meaning of content, to resolve this problem, we use stemming.

Stemming normalizes the word by truncating the word to its stem word. For example, the words “studies,” “studied,” “studying” will be reduced to **“studi,”** making all these word forms to refer to only one token. Notice that stemming may not give us a dictionary, grammatical word for a particular set of words.

Let’s take an example:

a. Porter’s Stemmer Example 1:

In the code snippet below, we show that all the words truncate to their stem words. However, notice that the stemmed word is not a dictionary word.



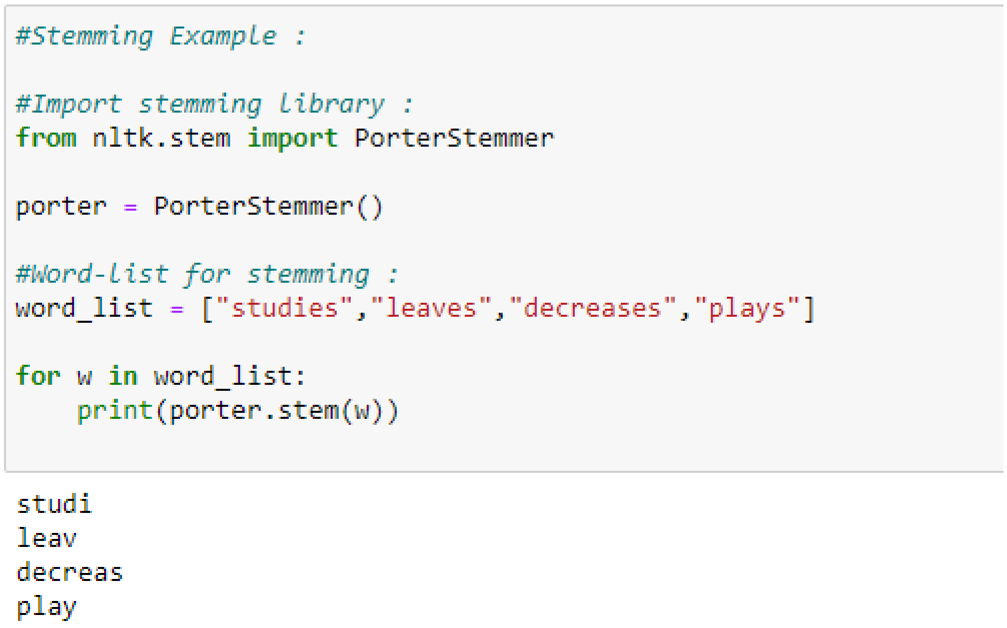
```
#Stemming Example :

#Import stemming library :
from nltk.stem import PorterStemmer

porter = PorterStemmer()

#Word-list for stemming :
word_list = ["Study","Studying","Studies","Studied"]

for w in word_list:
    print(porter.stem(w))
```

```
studi
studi
studi
studi
```

Figure 26: Code snippet showing a stemming example.

b. Porter's Stemmer Example 2:

In the code snippet below, many of the words after stemming did not end up being a recognizable dictionary word.



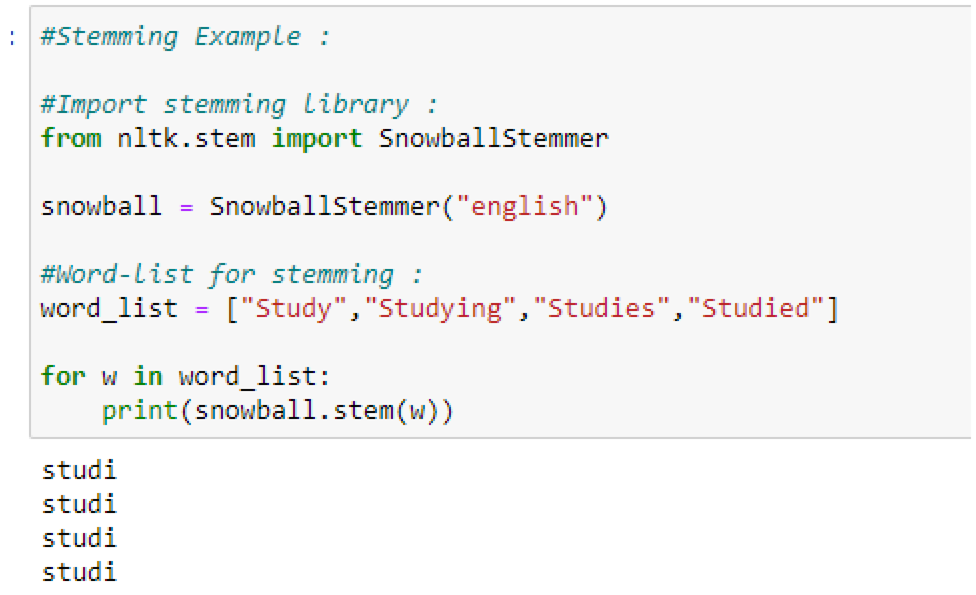
```
#Stemming Example :

#Import stemming library :
from nltk.stem import PorterStemmer

porter = PorterStemmer()

#Word-list for stemming :
word_list = ["studies","leaves","decreases","plays"]

for w in word_list:
    print(porter.stem(w))
```

```
studi
leav
decreas
play
```

Figure 27: Code snippet showing a stemming example.

c. SnowballStemmer:

SnowballStemmer generates the same output as porter stemmer, but it supports many more languages.

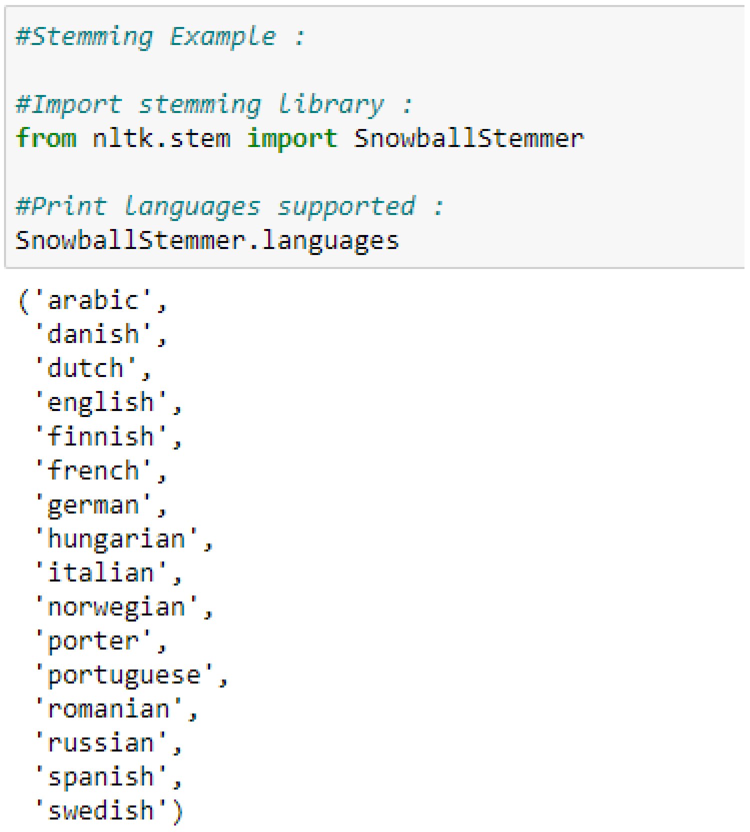


```
#Stemming Example :

#Import stemming library :
from nltk.stem import SnowballStemmer

snowball = SnowballStemmer("english")

#Word-List for stemming :
word_list = ["Study","Studying","Studies","Studied"]

for w in word_list:
    print(snowball.stem(w))
```

```
studi
studi
studi
studi
```

Figure 28: Code snippet showing an NLP stemming example.

d. Languages supported by snowball stemmer:



```
#Stemming Example :

#Import stemming library :
from nltk.stem import SnowballStemmer

#Print languages supported :
SnowballStemmer.languages
```

```
('arabic',
 'danish',
 'dutch',
 'english',
 'finnish',
 'french',
 'german',
 'hungarian',
 'italian',
 'norwegian',
 'porter',
 'portuguese',
 'romanian',
 'russian',
 'spanish',
 'swedish')
```

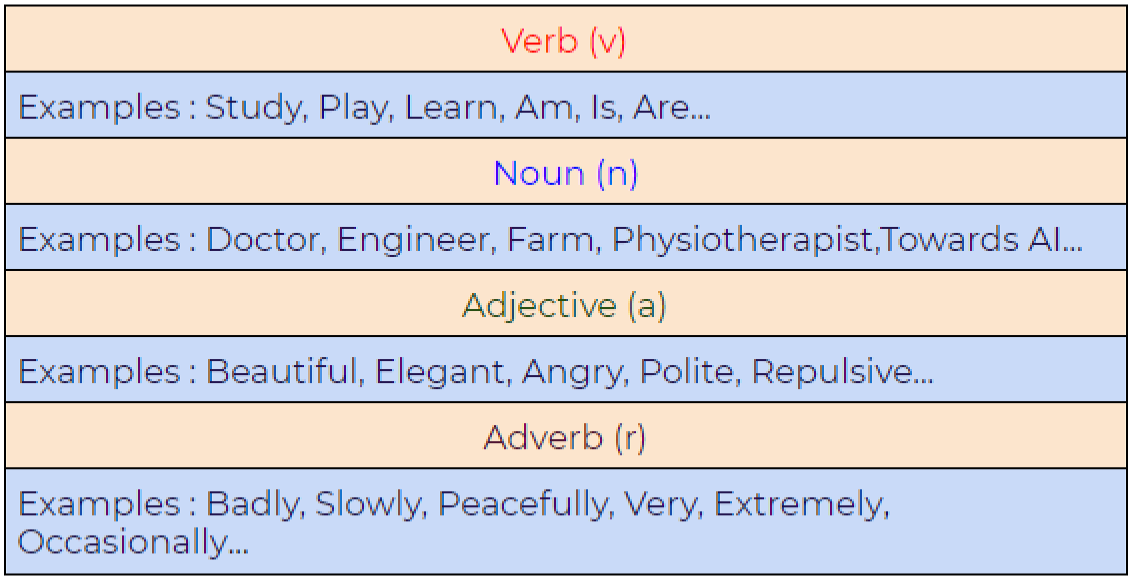
Figure 29: Code snippet showing an NLP stemming example.

Program 4: Lemmatization

Lemmatization tries to achieve a similar base "stem" for a word. However, what makes it different is that it finds the dictionary word instead of truncating the original word. Stemming does not consider the context of the word. That is why it generates results faster, but it is less accurate than lemmatization.

If accuracy is not the project's final goal, then stemming is an appropriate approach. If higher accuracy is crucial and the project is not on a tight deadline, then the best option is amortization (Lemmatization has a lower processing speed, compared to stemming).

Lemmatization takes into account Part Of Speech (POS) values. Also, lemmatization may generate different outputs for different values of POS. We generally have four choices for POS:



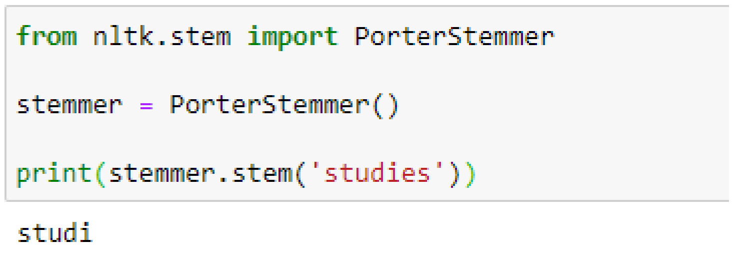
Verb (v)
Examples : Study, Play, Learn, Am, Is, Are...
Noun (n)
Examples : Doctor, Engineer, Farm, Physiotherapist,Towards AI...
Adjective (a)
Examples : Beautiful, Elegant, Angry, Polite, Repulsive...
Adverb (r)
Examples : Badly, Slowly, Peacefully, Very, Extremely, Occasionally...

Figure 36: Part of Speech (POS) values in lemmatization.

Difference between Stemmer and Lemmatizer:

a. Stemming:

Notice how on stemming, the word "studies" gets truncated to "studi."



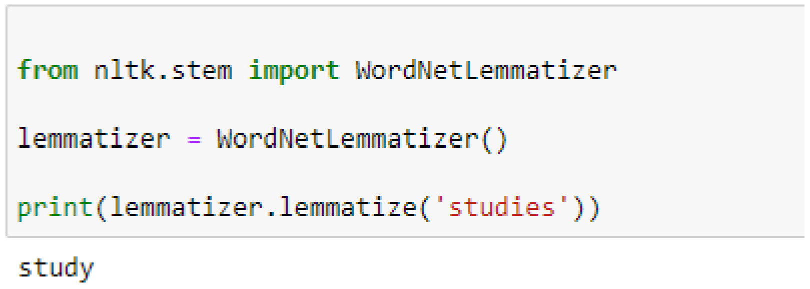
```python
from nltk.stem import PorterStemmer

stemmer = PorterStemmer()

print(stemmer.stem('studies'))
```

```
studi
```

Figure 47: Using stemming with the NLTK Python framework.

b. Lemmatizing:

During lemmatization, the word “studies” displays its dictionary word “study.”



```
from nltk.stem import WordNetLemmatizer

lemmatizer = WordNetLemmatizer()

print(lemmatizer.lemmatize('studies'))
```

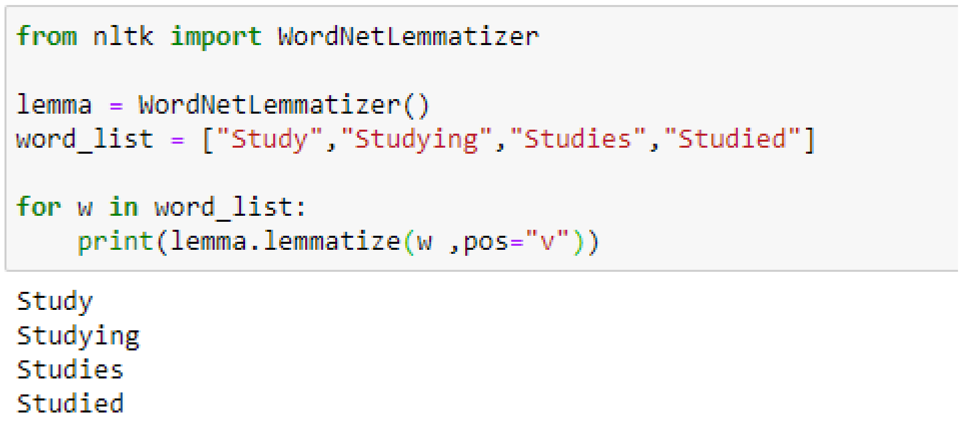
```
study
```

Figure 38: Using lemmatization with the NLTK Python framework.

Python Implementation:

a. A basic example demonstrating how a lemmatizer works

In the following example, we are taking the PoS tag as “verb,” and when we apply the lemmatization rules, it gives us dictionary words instead of truncating the original word:



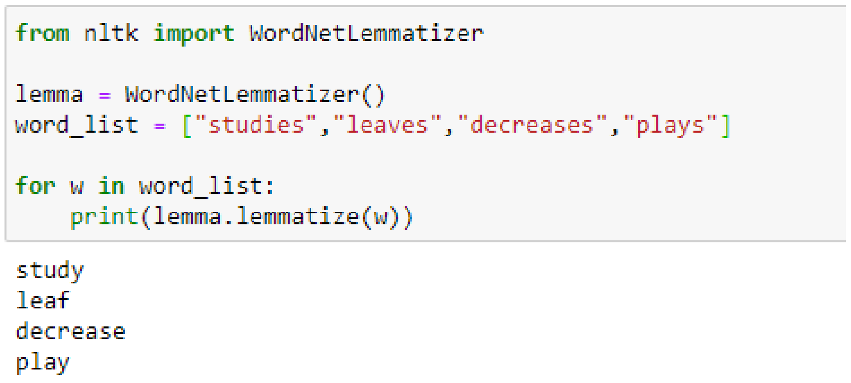
```
from nltk import WordNetLemmatizer

lemma = WordNetLemmatizer()
word_list = ["Study","Studying","Studies","Studied"]

for w in word_list:
    print(lemma.lemmatize(w ,pos="v"))
```

```
Study
Studying
Studies
Studied
```

Figure 39: Simple lemmatization example with the NLTK framework.

b. Lemmatizer with default PoS value

The default value of PoS in lemmatization is a noun(n). In the following example, we can see that it’s generating dictionary words:

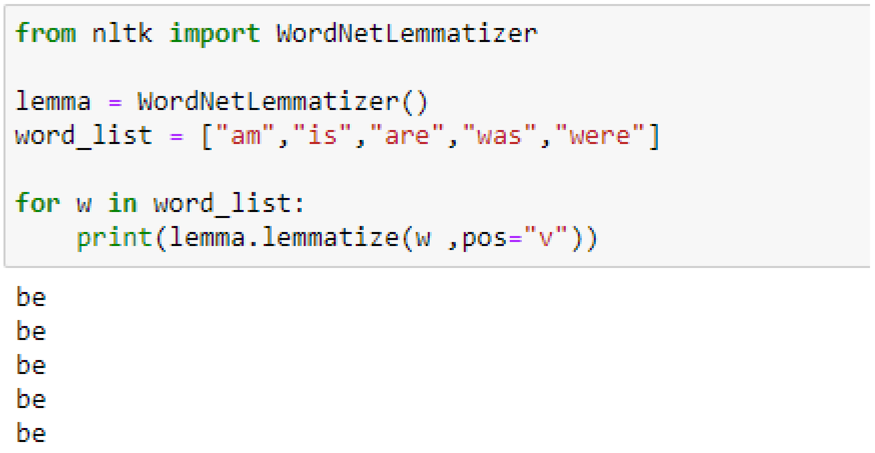


```python
from nltk import WordNetLemmatizer

lemma = WordNetLemmatizer()
word_list = ["studies","leaves","decreases","plays"]

for w in word_list:
    print(lemma.lemmatize(w))
```

```
study
leaf
decrease
play
```

Figure 40: Using lemmatization to generate default values.

c. Another example demonstrating the power of lemmatizer

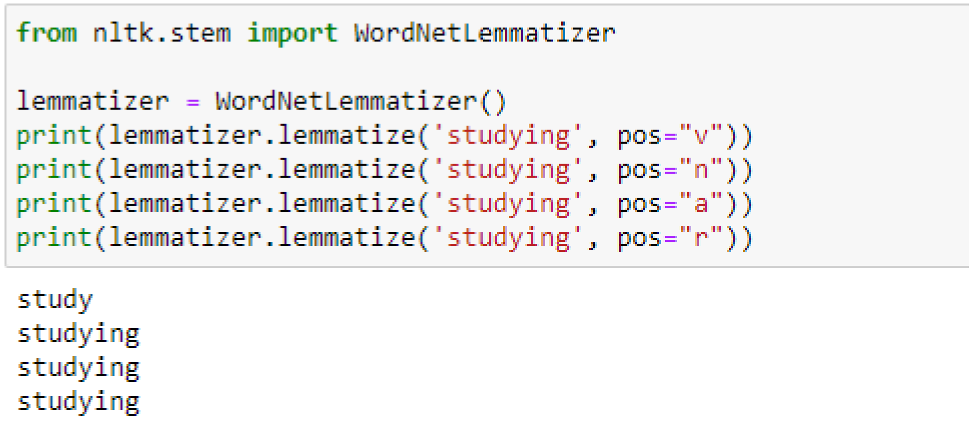


```python
from nltk import WordNetLemmatizer

lemma = WordNetLemmatizer()
word_list = ["am","is","are","was","were"]

for w in word_list:
    print(lemma.lemmatize(w ,pos="v"))
```

```
be
be
be
be
be
```

Figure 41: Lemmatization of the words: “am”, “are”, “is”, “was”, “were”

d. Lemmatizer with different POS values



```python
from nltk.stem import WordNetLemmatizer

lemmatizer = WordNetLemmatizer()
print(lemmatizer.lemmatize('studying', pos="v"))
print(lemmatizer.lemmatize('studying', pos="n"))
print(lemmatizer.lemmatize('studying', pos="a"))
print(lemmatizer.lemmatize('studying', pos="r"))
```

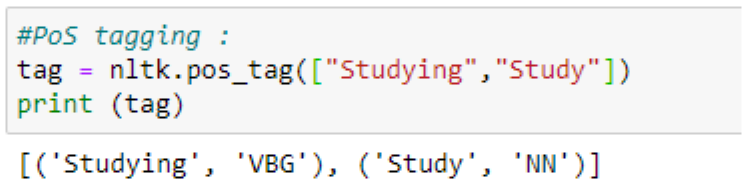
```
study
studying
studying
studying
```

Figure 42: Lemmatization with different Part-of-Speech values

Program 5: PoS Tagging with all PoS Tags

Python Implementation:

a. A simple example demonstrating PoS tagging.

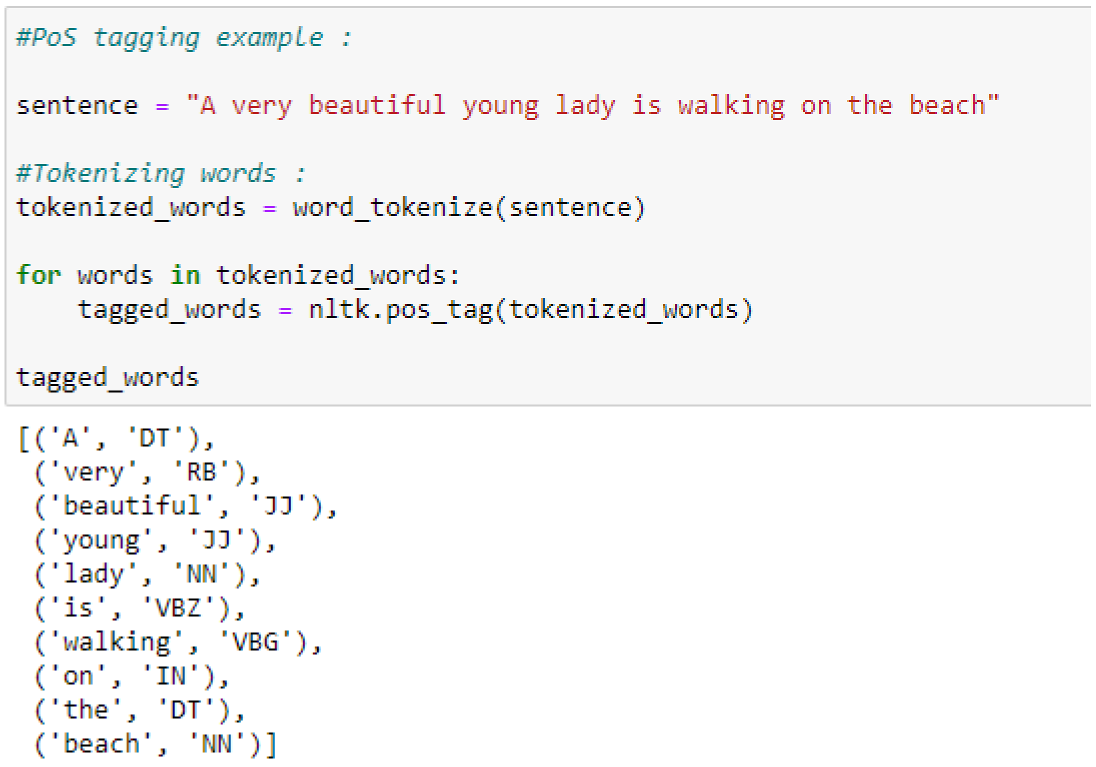


```
#PoS tagging :
tag = nltk.pos_tag(["Studying","Study"])
print (tag)
```

```
[('Studying', 'VBG'), ('Study', 'NN')]
```

Figure 79: PoS tagging example.

b. A full example demonstrating the use of PoS tagging.



```
#PoS tagging example :

sentence = "A very beautiful young lady is walking on the beach"

#Tokenizing words :
tokenized_words = word_tokenize(sentence)

for words in tokenized_words:
    tagged_words = nltk.pos_tag(tokenized_words)

tagged_words
```

```
[('A', 'DT'),
 ('very', 'RB'),
 ('beautiful', 'JJ'),
 ('young', 'JJ'),
 ('lady', 'NN'),
 ('is', 'VBZ'),
 ('walking', 'VBG'),
 ('on', 'IN'),
 ('the', 'DT'),
 ('beach', 'NN')]
```

Figure 90: Full Python sample demonstrating PoS tagging.

Program 6: Chunking Process

Chunking means to extract meaningful phrases from unstructured text. By tokenizing a book into words, it’s sometimes hard to infer meaningful information. It works on top of Part of Speech(PoS) tagging. Chunking takes PoS tags as input and provides chunks as output. Chunking literally means a group of words, which breaks simple text into phrases that are more meaningful than individual words.

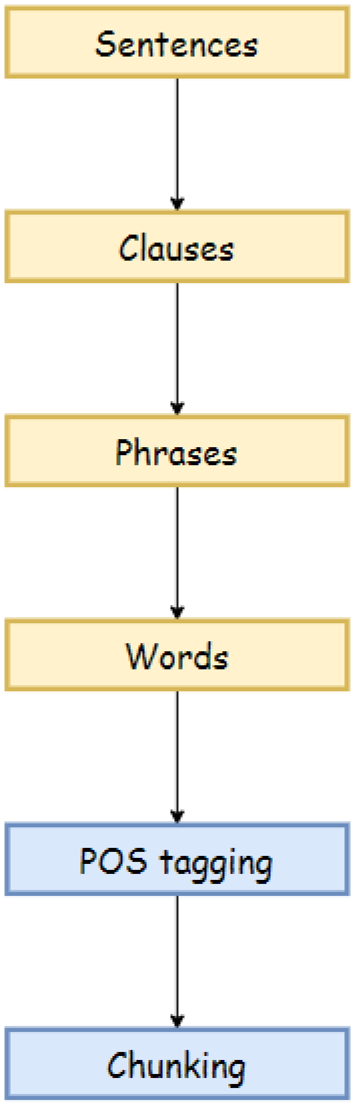


Figure 81: The chunking process in NLP.

Before working with an example, we need to know what phrases are? Meaningful groups of words are called phrases. There are five significant categories of phrases.

Noun Phrases (NP).

Verb Phrases (VP).

Adjective Phrases (ADJP).

Adverb Phrases (ADVP).

Prepositional Phrases (PP).

Phrase structure rules:

S(Sentence) → NP VP.

NP → {Determiner, Noun, Pronoun, Proper name}.

VP → V (NP)(PP)(Adverb).

PP → Pronoun (NP).

AP → Adjective (PP).

Example:

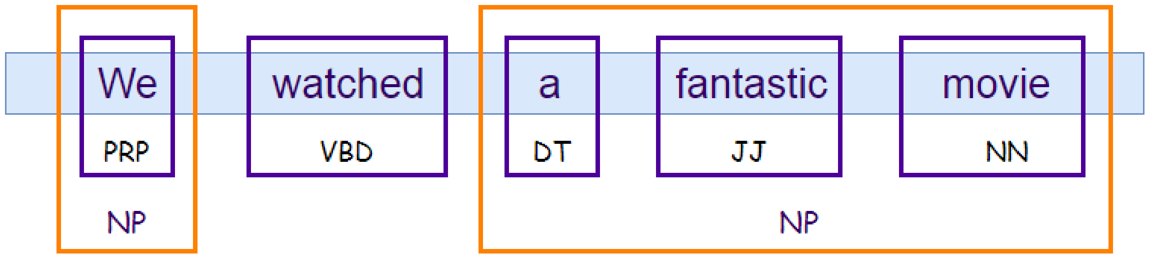


Figure 82: A chunking example in NLP.

Python Implementation:

In the following example, we will extract a noun phrase from the text. Before extracting it, we need to define what kind of noun phrase we are looking for, or in other words, we have to set the grammar for a noun phrase. In this case, we define a noun phrase by an optional determiner followed by adjectives and nouns. Then we can define other rules to extract some other phrases. Next, we are going to use RegexpParser( ) to parse the grammar. Notice that we can also visualize the text with the .draw( ) function.

```
#Extracting Noun Phrase from text :

# ? - optional character
# * - 0 or more repetitions
grammar = "NP : {<DT>?<JJ>*<NN>} "

#Creating a parser :
parser = nltk.RegexpParser(grammar)

#Parsing text :
output = parser.parse(tagged_words)
print (output)

#To visualize :
output.draw()
```

```
(S
  A/DT
  very/RB
  (NP beautiful/JJ young/JJ lady/NN)
  is/VBZ
  walking/VBG
  on/IN
  (NP the/DT beach/NN))
```

Figure 83: Code snippet to extract noun phrases from a text file.

In this example, we can see that we have successfully extracted the noun phrase from the text.

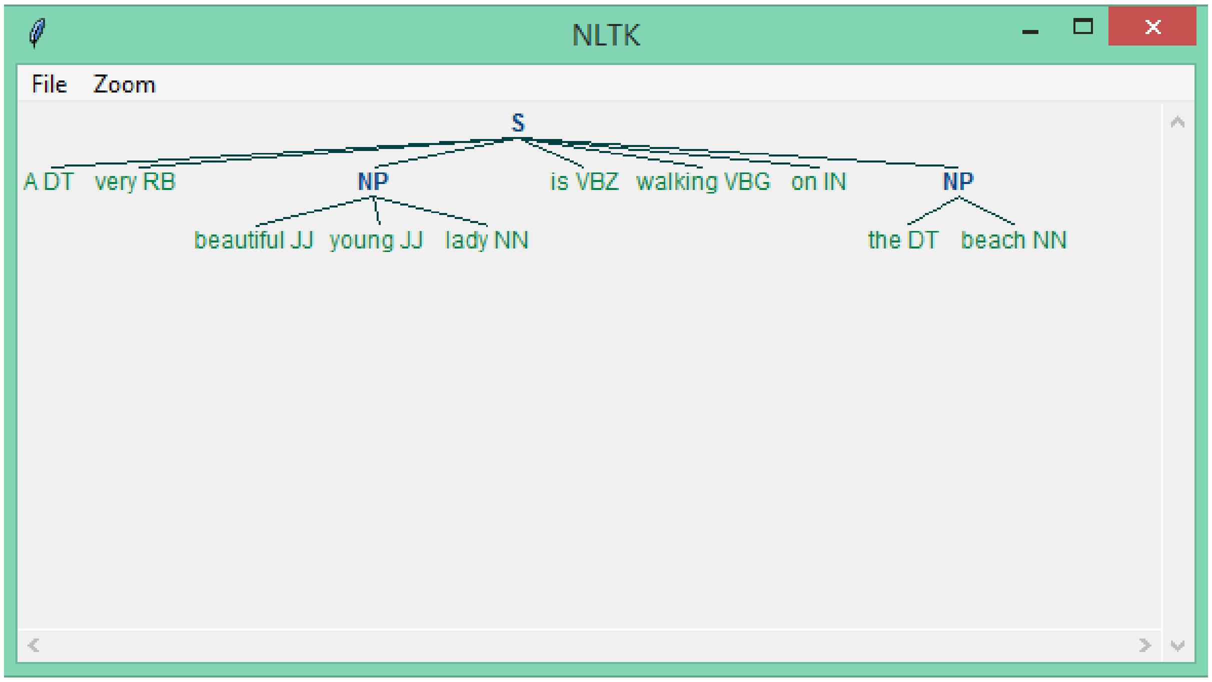


Figure 84: Successful extraction of the noun phrase from the input text.

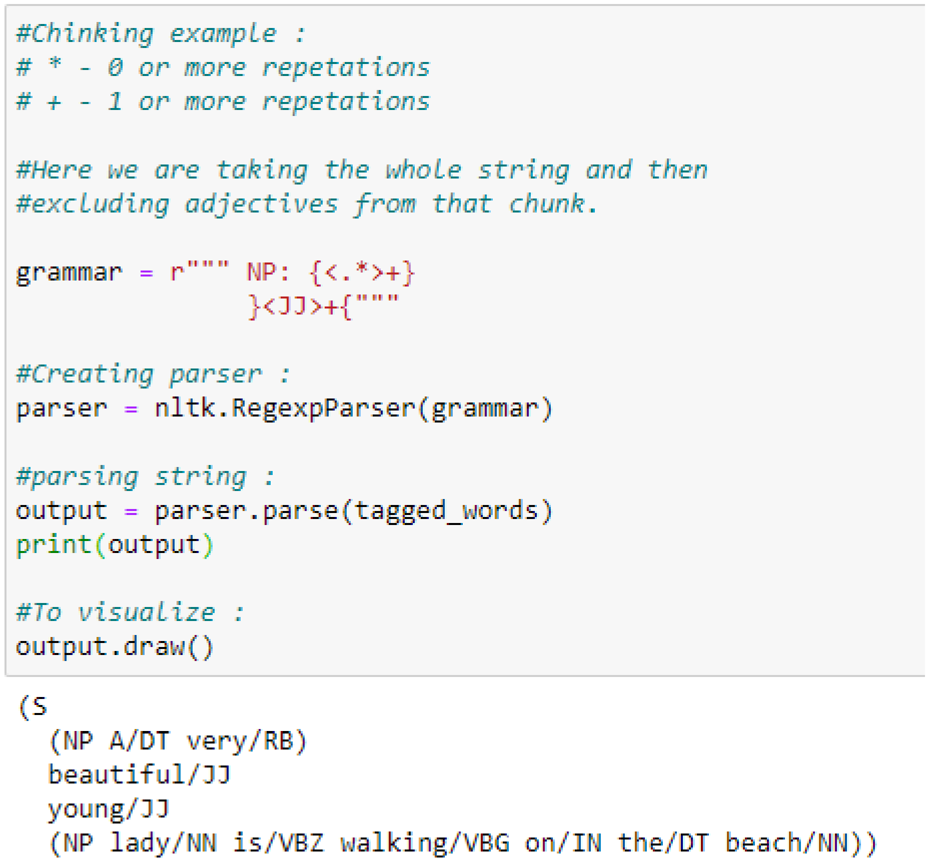
Program 7: Chinking Process

Chinking excludes a part from our chunk. There are certain situations where we need to exclude a part of the text from the whole text or chunk. In complex extractions, it is possible that chunking can output unuseful data. In such case scenarios, we can use chinking to exclude some parts from that chunked text.

In the following example, we are going to take the whole string as a chunk, and then we are going to exclude adjectives from it by using chinking. We generally use chinking when we have a lot of un useful data even after chunking. Hence, by using this method, we can easily set that apart, also to write chinking grammar, we have to use inverted curly braces, i.e.:

} **write chinking grammar here** {

Python Implementation:



```
#Chinking example :
# * - 0 or more repetations
# + - 1 or more repetations

#Here we are taking the whole string and then
#excluding adjectives from that chunk.

grammar = r""" NP: {<.*>+}
               }<JJ>+{"""

#Creating parser :
parser = nltk.RegexpParser(grammar)

#parsing string :
output = parser.parse(tagged_words)
print(output)

#To visualize :
output.draw()
```

```
(S
  (NP A/DT very/RB)
  beautiful/JJ
  young/JJ
  (NP lady/NN is/VBZ walking/VBG on/IN the/DT beach/NN))
```

Figure 85: Chinking implementation with Python.

From the example above, we can see that adjectives separate from the other text.

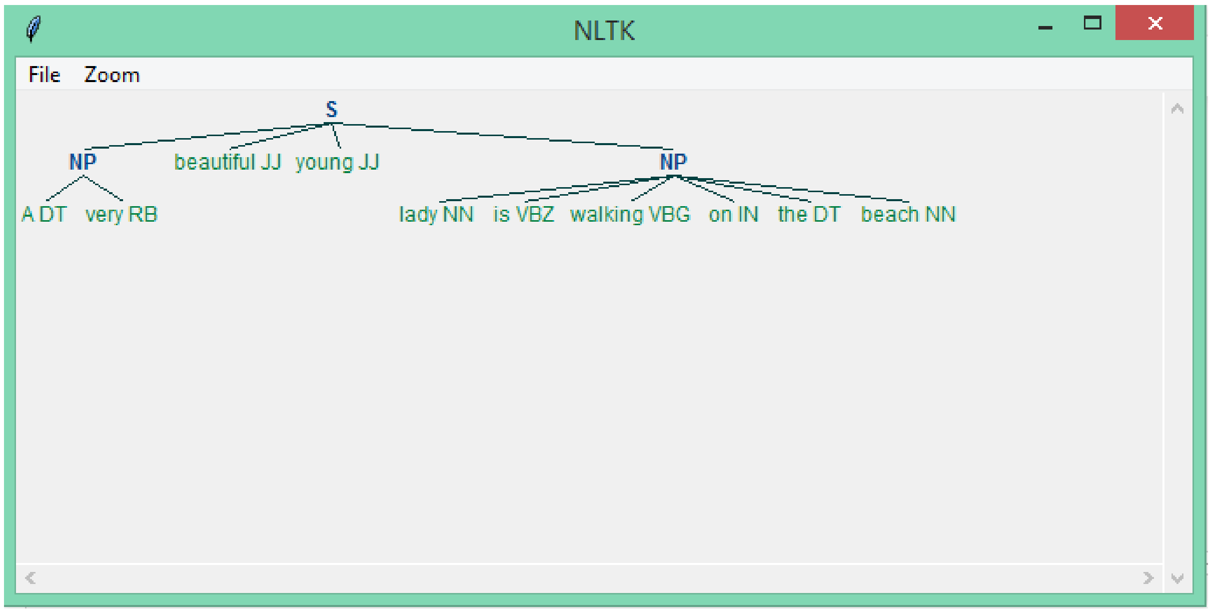


Figure 86: In this example, adjectives are excluded by using chinking.

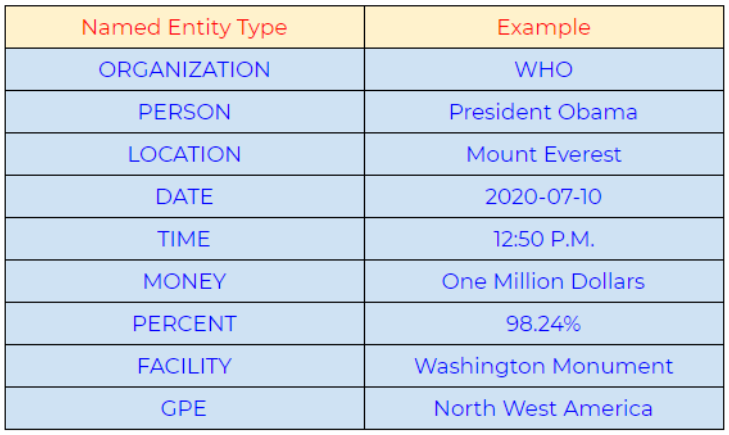
Program 8: Named Entity Recognition

Named entity recognition can automatically scan entire articles and pull out some fundamental entities like people, organizations, places, date, time, money, and GPE discussed in them.

Use-Cases:

1. Content classification for news channels.
2. Summarizing resumes.
3. Optimizing search engine algorithms.
4. Recommendation Systems
5. Customer support.

Commonly used types of named entity:



Named Entity Type	Example
ORGANIZATION	WHO
PERSON	President Obama
LOCATION	Mount Everest
DATE	2020-07-10
TIME	12:50 P.M.
MONEY	One Million Dollars
PERCENT	98.24%
FACILITY	Washington Monument
GPE	North West America

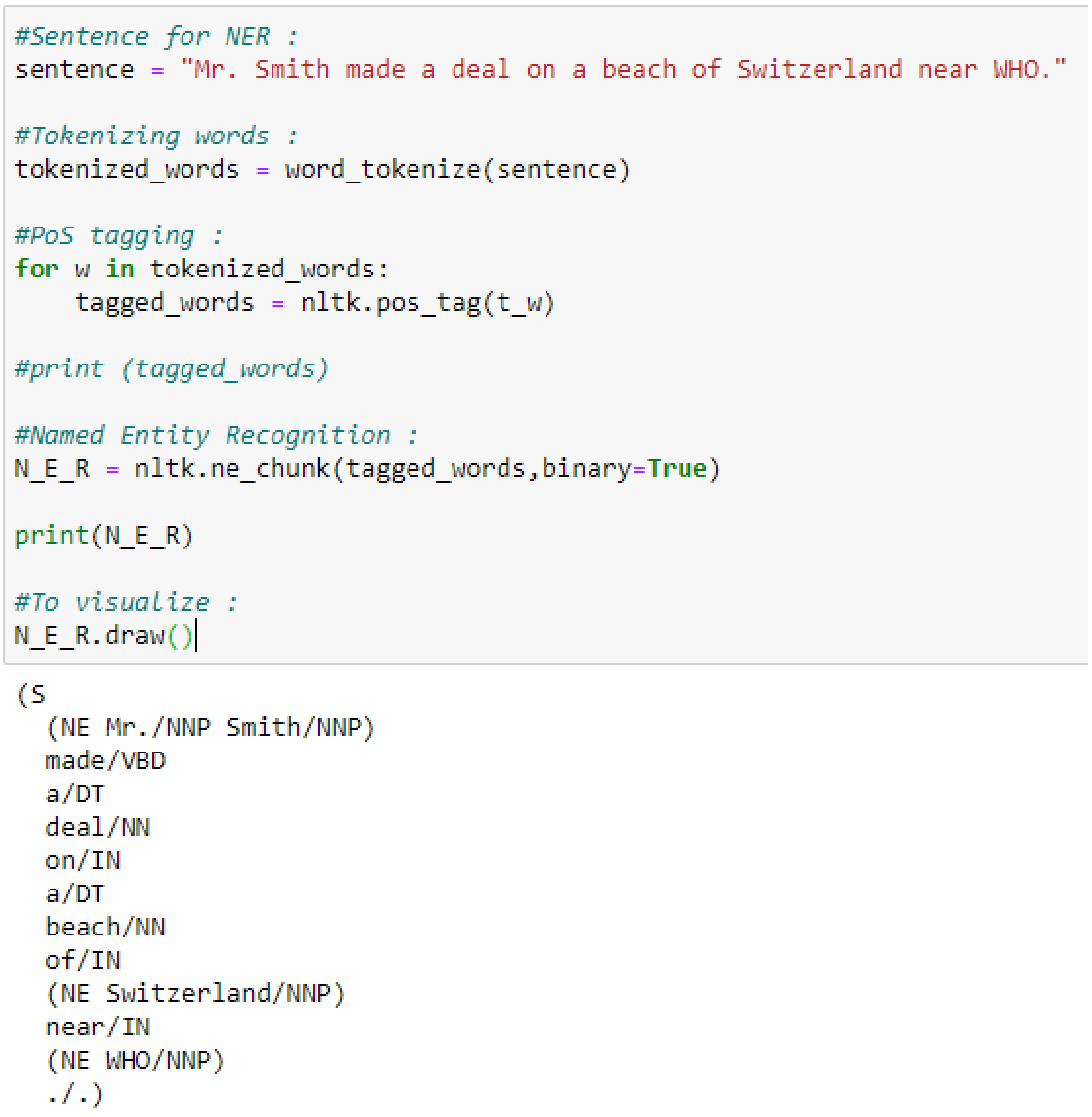
Figure 87: An example of commonly used types of named entity recognition (NER).

Python Implementation:

There are two options :

1. binary = True

When the binary value is True, then it will only show whether a particular entity is named entity or not. It will not show any further details on it.



```
#Sentence for NER :
sentence = "Mr. Smith made a deal on a beach of Switzerland near WHO."

#Tokenizing words :
tokenized_words = word_tokenize(sentence)

#PoS tagging :
for w in tokenized_words:
    tagged_words = nltk.pos_tag(t_w)

#print (tagged_words)

#Named Entity Recognition :
N_E_R = nltk.ne_chunk(tagged_words,binary=True)

print(N_E_R)

#To visualize :
N_E_R.draw()
```

```
(S
  (NE Mr./NNP Smith/NNP)
  made/VBD
  a/DT
  deal/NN
  on/IN
  a/DT
  beach/NN
  of/IN
  (NE Switzerland/NNP)
  near/IN
  (NE WHO/NNP)
  ./.)
```

Figure 88: Python implementation when a binary value is True.

Our graph does not show what type of named entity it is. It only shows whether a particular word is named entity or not.

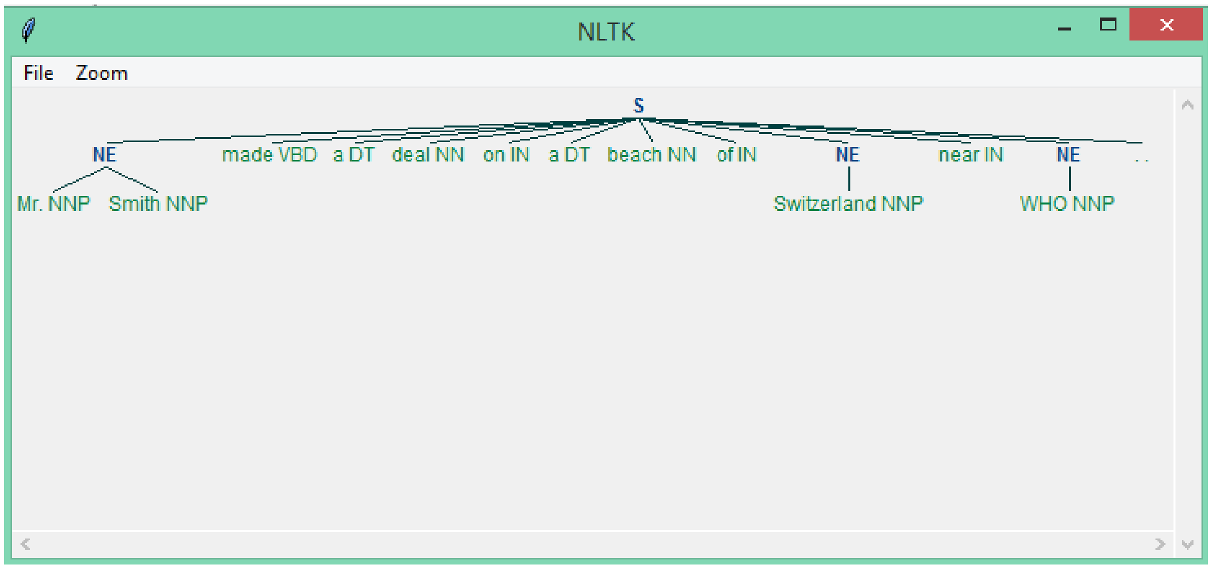


Figure 89: Graph example of when a binary value is True.

2. binary = False

When the binary value equals False, it shows in detail the type of named entities.

```
#Sentence for NER :
sentence = "Mr. Smith made a deal on a beach of Switzerland near WHO."

#Tokenizing words :
tokenized_words = word_tokenize(sentence)

#PoS tagging :
for w in tokenized_words:
    tagged_words = nltk.pos_tag(t_w)

#print (tagged_words)

#Named Entity Recognition :
N_E_R = nltk.ne_chunk(tagged_words,binary=False)
print(N_E_R)

#To visualize :
N_E_R.draw()
```

```
(S
  (PERSON Mr./NNP)
  (PERSON Smith/NNP)
  made/VBD
  a/DT
  deal/NN
  on/IN
  a/DT
  beach/NN
  of/IN
  (GPE Switzerland/NNP)
  near/IN
  (ORGANIZATION WHO/NNP)
  ./.)
```

Figure 90: Python implementation when a binary value is False.

Our graph now shows what type of named entity it is.

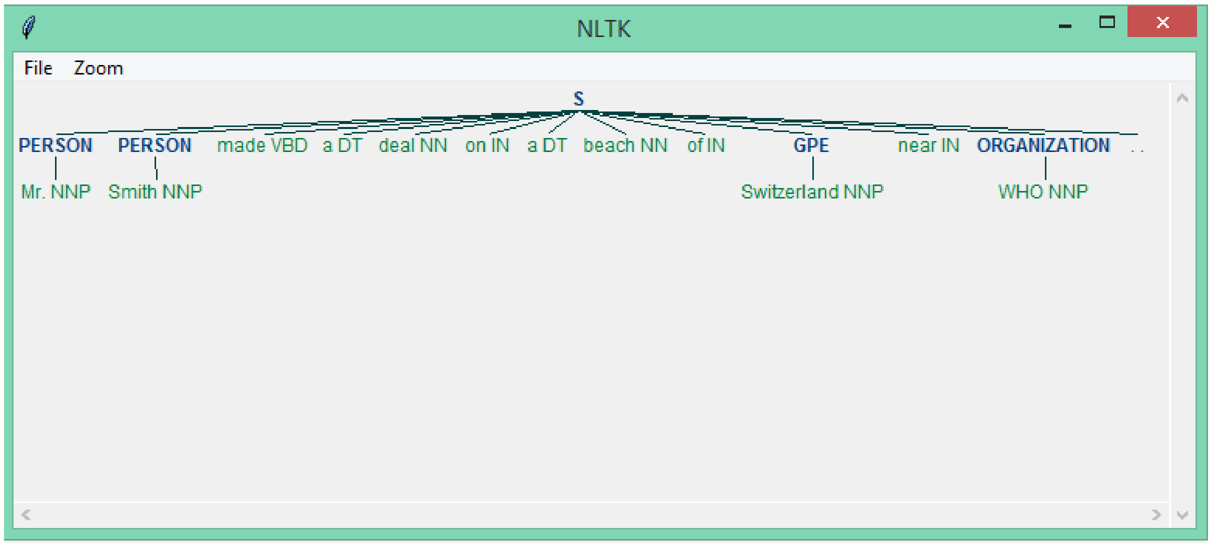


Figure 91: Graph showing the type of named entities when a binary value equals false.

Program 9: Wordnet

Wordnet is a lexical database for the English language. Wordnet is a part of the NLTK corpus. We can use Wordnet to find meanings of words, synonyms, antonyms, and many other words.

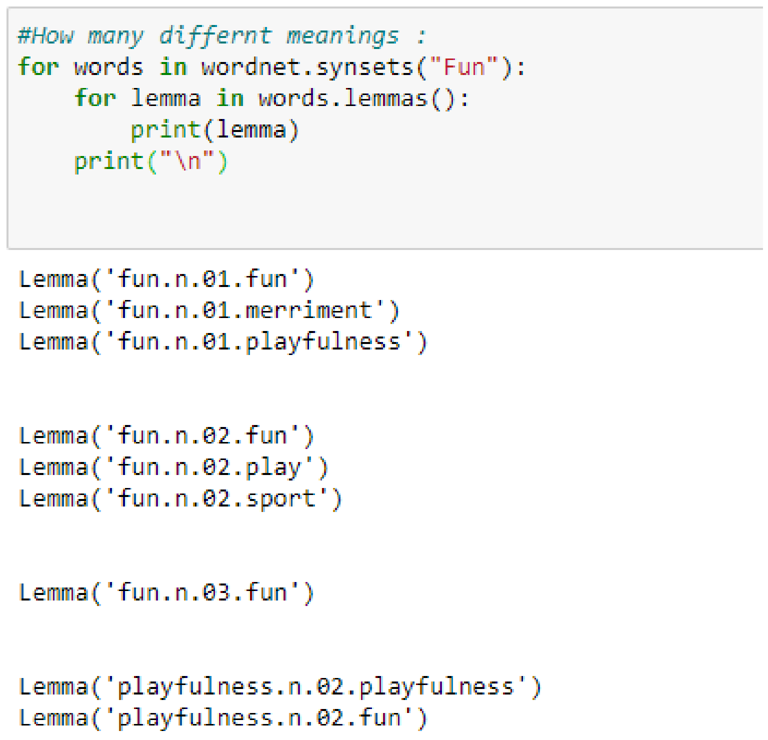
a. We can check how many different definitions of a word are available in Wordnet.

```
from nltk.corpus import wordnet

for words in wordnet.synsets("Fun"):
    print(words)
```

```
Synset('fun.n.01')
Synset('fun.n.02')
Synset('fun.n.03')
Synset('playfulness.n.02')
```

Figure 92: Checking word definitions with Wordnet using the NLTK framework.

b. We can also check the meaning of those different definitions.

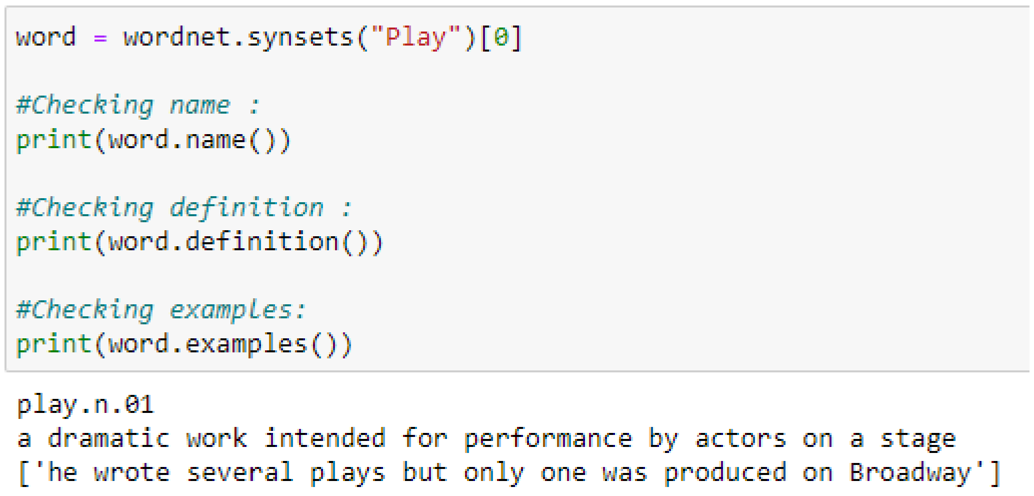


```
#How many differnt meanings :
for words in wordnet.synsets("Fun"):
    for lemma in words.lemmas():
        print(lemma)
    print("\n")
```

```
Lemma('fun.n.01.fun')
Lemma('fun.n.01.merriment')
Lemma('fun.n.01.playfulness')


Lemma('fun.n.02.fun')
Lemma('fun.n.02.play')
Lemma('fun.n.02.sport')


Lemma('fun.n.03.fun')


Lemma('playfulness.n.02.playfulness')
Lemma('playfulness.n.02.fun')
```

Figure 93: Gathering the meaning of the different definitions by using Wordnet.

c. All details for a word.

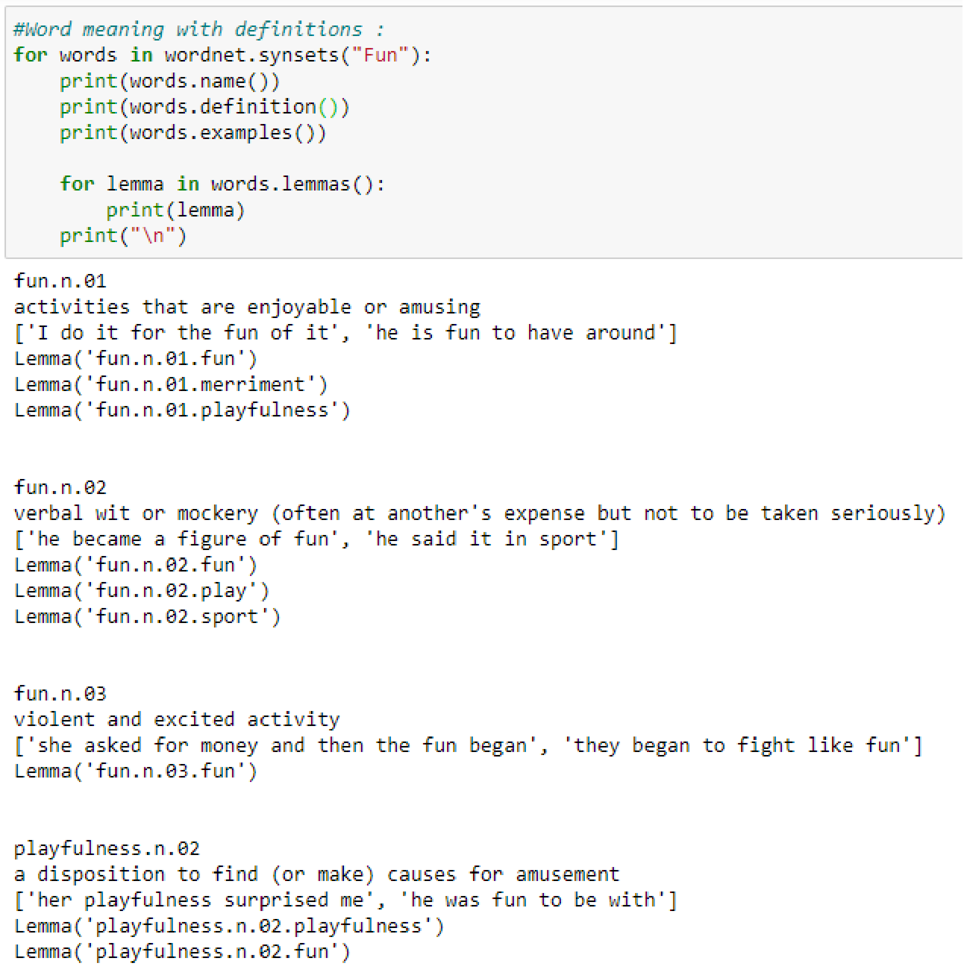


```
word = wordnet.synsets("Play")[0]

#Checking name :
print(word.name())

#Checking definition :
print(word.definition())

#Checking examples:
print(word.examples())
```

```
play.n.01
a dramatic work intended for performance by actors on a stage
['he wrote several plays but only one was produced on Broadway']
```

Figure: 94: Finding all the details for a specific word.

d. All details for all meanings of a word.

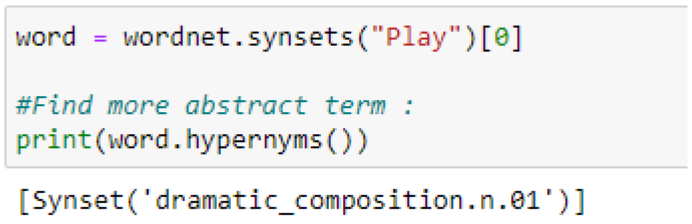


```
#Word meaning with definitions :
for words in wordnet.synsets("Fun"):
    print(words.name())
    print(words.definition())
    print(words.examples())

    for lemma in words.lemmas():
        print(lemma)
    print("\n")
```

```
fun.n.01
activities that are enjoyable or amusing
['I do it for the fun of it', 'he is fun to have around']
Lemma('fun.n.01.fun')
Lemma('fun.n.01.merriment')
Lemma('fun.n.01.playfulness')


fun.n.02
verbal wit or mockery (often at another's expense but not to be taken seriously)
['he became a figure of fun', 'he said it in sport']
Lemma('fun.n.02.fun')
Lemma('fun.n.02.play')
Lemma('fun.n.02.sport')


fun.n.03
violent and excited activity
['she asked for money and then the fun began', 'they began to fight like fun']
Lemma('fun.n.03.fun')


playfulness.n.02
a disposition to find (or make) causes for amusement
['her playfulness surprised me', 'he was fun to be with']
Lemma('playfulness.n.02.playfulness')
Lemma('playfulness.n.02.fun')
```

Figure 95: Finding all details for all the meanings of a specific word.

e. Hypernyms: Hypernyms gives us a more abstract term for a word.

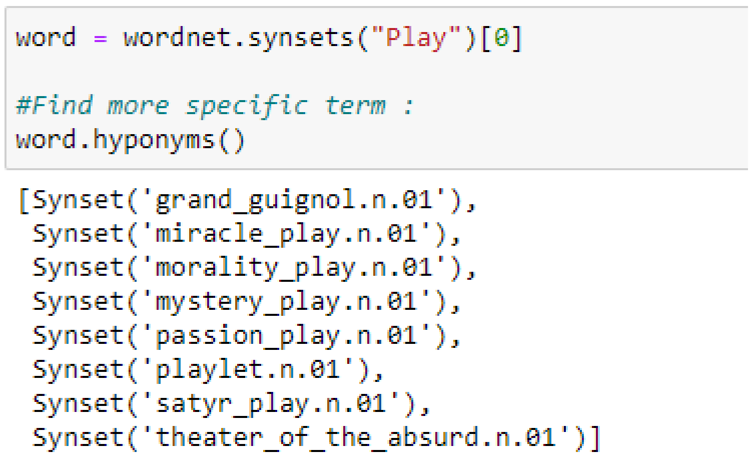


```
word = wordnet.synsets("Play")[0]

#Find more abstract term :
print(word.hypernyms())
```

```
[Synset('dramatic_composition.n.01')]
```

Figure 96: Using Wordnet to find a hypernym.

f. Hyponyms: Hyponyms gives us a more specific term for a word.

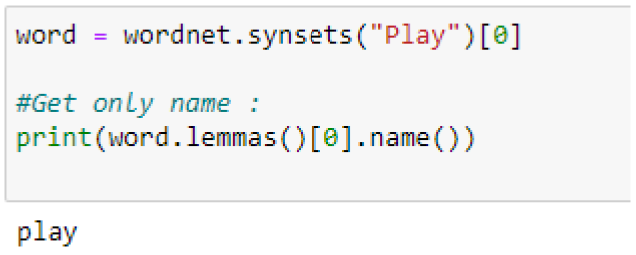


```
word = wordnet.synsets("Play")[0]

#Find more specific term :
word.hyponyms()
```

```
[Synset('grand_guignol.n.01'),
 Synset('miracle_play.n.01'),
 Synset('morality_play.n.01'),
 Synset('mystery_play.n.01'),
 Synset('passion_play.n.01'),
 Synset('playlet.n.01'),
 Synset('satyr_play.n.01'),
 Synset('theater_of_the_absurd.n.01')]
```

Figure 97: Using Wordnet to find a hyponym.

g. Get a name only.

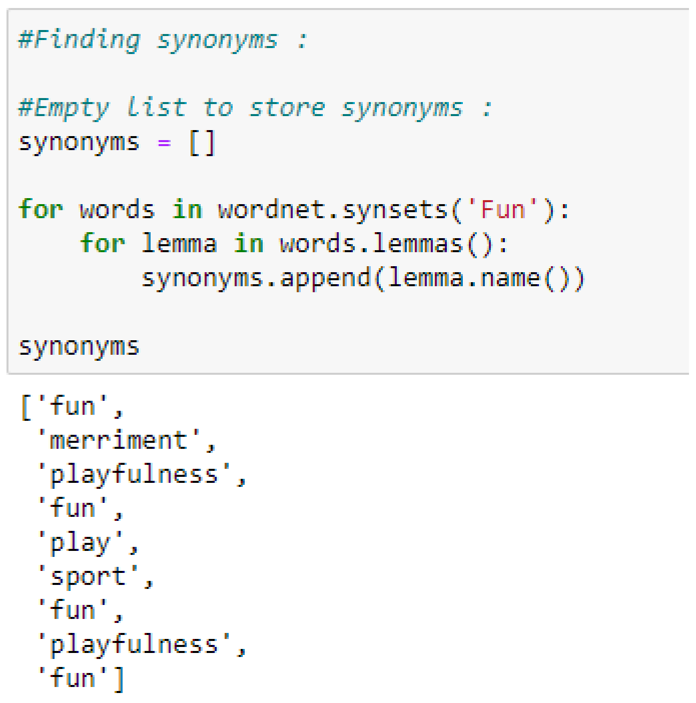


```
word = wordnet.synsets("Play")[0]

#Get only name :
print(word.lemmas()[0].name())
```

```
play
```

Figure 98: Finding only a name with Wordnet.

h. Synonyms.

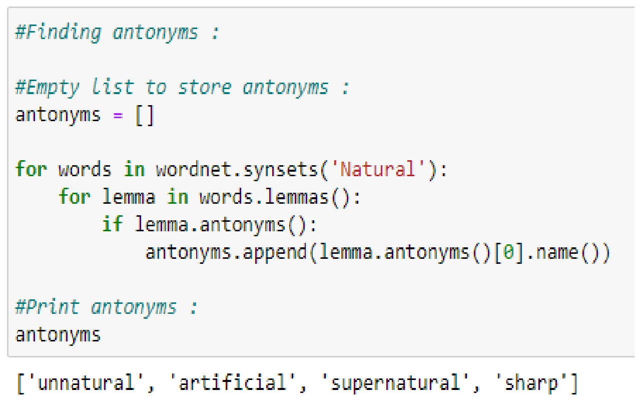


```
#Finding synonyms :

#Empty list to store synonyms :
synonyms = []

for words in wordnet.synsets('Fun'):
    for lemma in words.lemmas():
        synonyms.append(lemma.name())

synonyms
```

```
['fun',
 'merriment',
 'playfulness',
 'fun',
 'play',
 'sport',
 'fun',
 'playfulness',
 'fun']
```

Figure 99: Finding synonyms with Wordnet.

i. Antonyms.

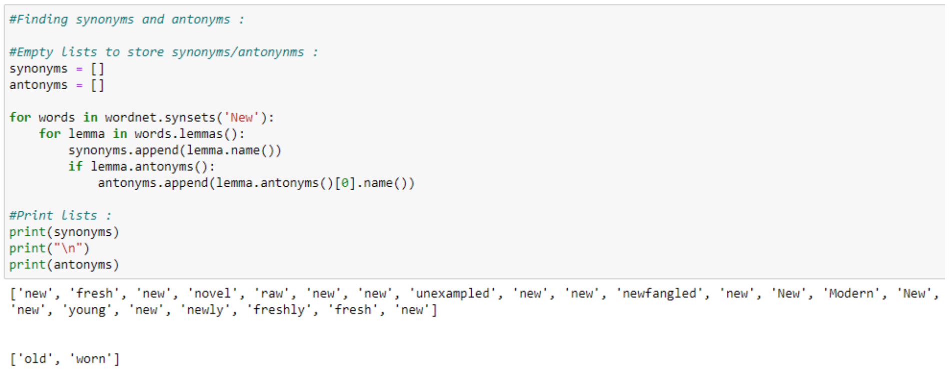


```
#Finding antonyms :

#Empty list to store antonyms :
antonyms = []

for words in wordnet.synsets('Natural'):
    for lemma in words.lemmas():
        if lemma.antonyms():
            antonyms.append(lemma.antonyms()[0].name())

#Print antonyms :
antonyms
```

```
['unnatural', 'artificial', 'supernatural', 'sharp']
```

Figure 100: Finding antonyms with Wordnet.

j. Synonyms and antonyms.

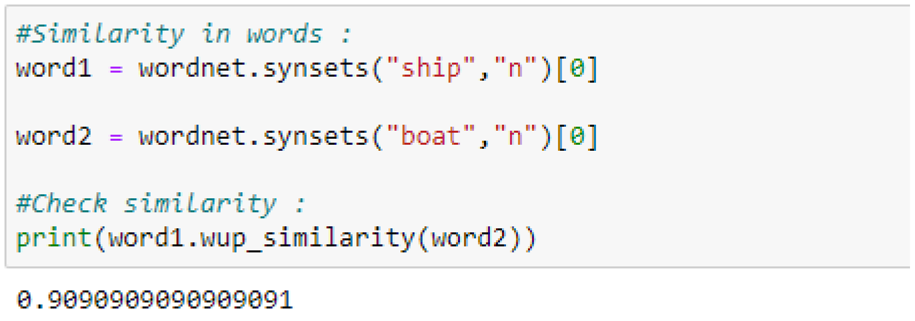


```
#Finding synonyms and antonyms :

#Empty lists to store synonyms/antonynms :
synonyms = []
antonyms = []

for words in wordnet.synsets('New'):
    for lemma in words.lemmas():
        synonyms.append(lemma.name())
        if lemma.antonyms():
            antonyms.append(lemma.antonyms()[0].name())

#Print lists :
print(synonyms)
print("\n")
print(antonyms)
```

```
['new', 'fresh', 'new', 'novel', 'raw', 'new', 'new', 'unexampled', 'new', 'new', 'newfangled', 'new', 'New', 'Modern', 'New',
'new', 'young', 'new', 'newly', 'freshly', 'fresh', 'new']


['old', 'worn']
```

Figure 101: Finding synonyms and antonyms code snippet with Wordnet.

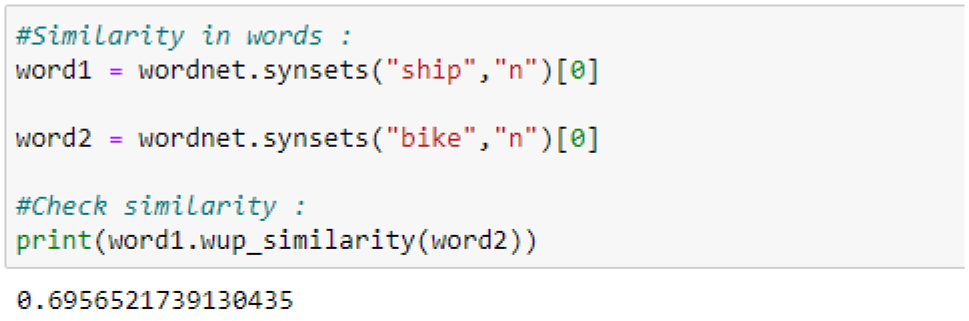
k. Finding the similarity between words.



```
#Similarity in words :
word1 = wordnet.synsets("ship","n")[0]

word2 = wordnet.synsets("boat","n")[0]

#Check similarity :
print(word1.wup_similarity(word2))
```

```
0.9090909090909091
```

Figure 102: Finding the similarity ratio between words using Wordnet.



```
#Similarity in words :
word1 = wordnet.synsets("ship","n")[0]

word2 = wordnet.synsets("bike","n")[0]

#Check similarity :
print(word1.wup_similarity(word2))
```

```
0.6956521739130435
```

Figure 103: Finding the similarity ratio between words using Wordnet.

Program 10: Bag of Words

Figure 104: A representation of a bag of words.

What is the Bag-of-Words method?

It is a method of extracting essential features from row text so that we can use it for machine learning models. We call it **“Bag”** of words because we discard the order of occurrences of words. A bag of words model converts the raw text into words, and it also counts the frequency for the words in the text. In summary, a bag of words is a collection of words that represent a sentence along with the word count where the order of occurrences is not relevant.

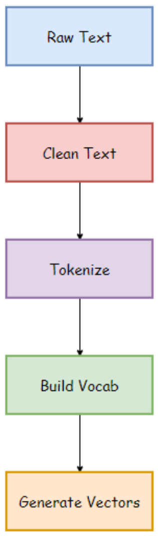


Figure 105: Structure of a bag of words.

Raw Text: This is the original text on which we want to perform analysis.

Clean Text: Since our raw text contains some unnecessary data like punctuation marks and stopwords, so we need to clean up our text. Clean text is the text after removing such words.

Tokenize: Tokenization represents the sentence as a group of tokens or words.

Building Vocab: It contains total words used in the text after removing unnecessary data.

Generate Vocab: It contains the words along with their frequencies in the sentences.

For instance:

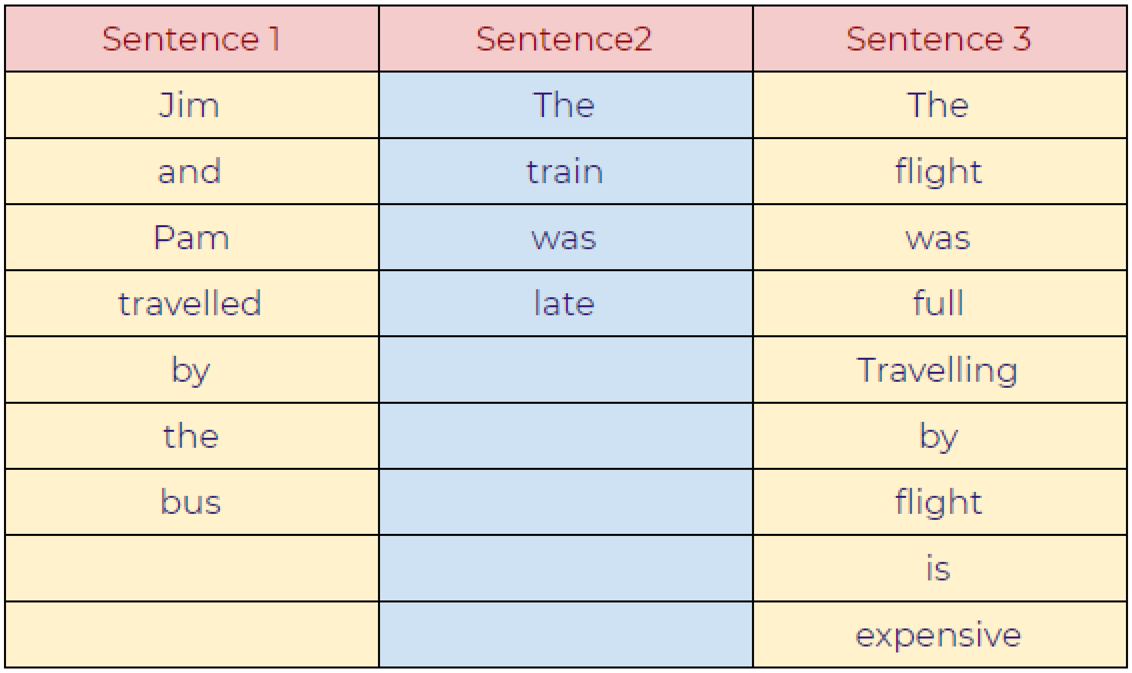
Sentences:

Jim and Pam traveled by bus.

The train was late.

The flight was full. Traveling by flight is expensive.

a. Creating a basic structure:



Sentence 1	Sentence2	Sentence 3
Jim	The	The
and	train	flight
Pam	was	was
travelled	late	full
by		Travelling
the		by
bus		flight
		is
		expensive

Figure 106: Example of a basic structure for a bag of words.

b. Words with frequencies:

Sentence1	Count	Sentence2	Count	Sentence3	Count
Jim	1	The	1	The	1
and	1	train	1	flight	2
Pam	1	was	1	was	1
travelled	1	late	1	full	1
by	1			Travelling	1
the	1			by	1
bus	1			is	1
				expensive	1

Figure 107: Example of a basic structure for words with frequencies.

c. Combining all the words:

Sentence	Frequency
and	1
bus	1
by	2
expensive	1
flight	2
full	1
is	1
jim	1
late	1
pam	1
the	3
train	1
travelled	1
travelling	1
was	1

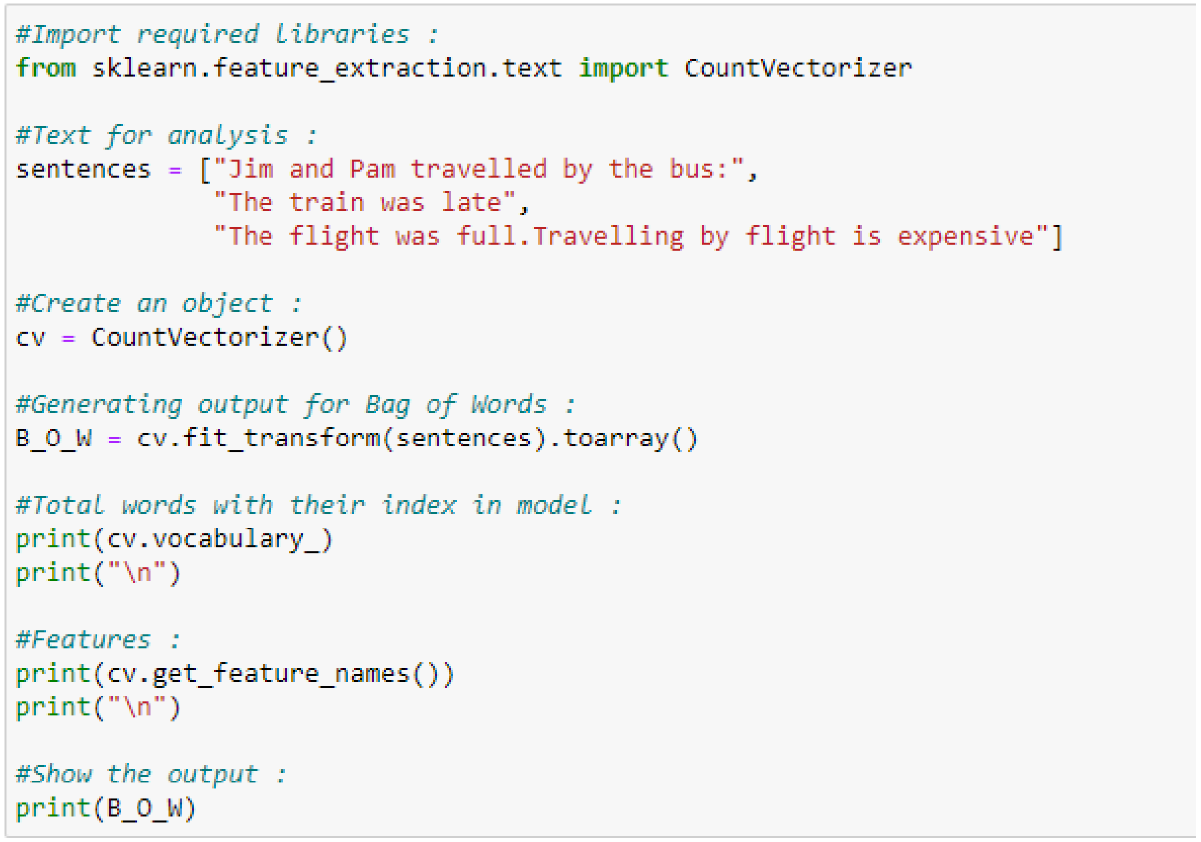
Figure 108: Combination of all the input words.

d. Final model:

	and	bus	by	expensive	flight	full	is	jim	Late	pam	The	train	travelled	travelling	was
S-1	1	1	1	0	0	0	0	1	0	1	1	0	1	0	0
S-2	0	0	0	0	0	0	0	0	1	0	1	1	0	0	1
S-3	0	0	1	1	2	1	1	0	0	0	1	0	0	1	1

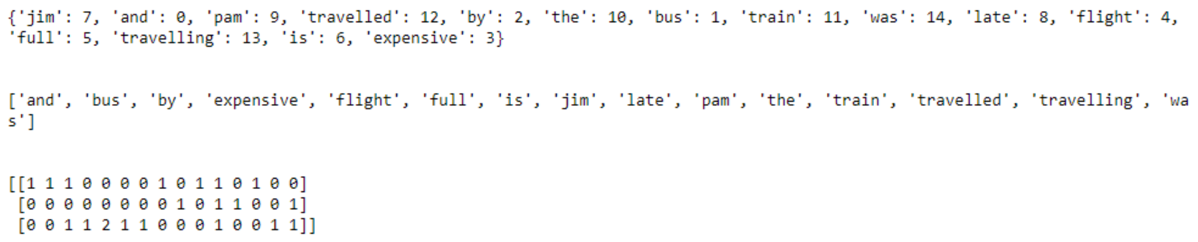
Figure 109: The final model of our bag of words.

Python Implementation:



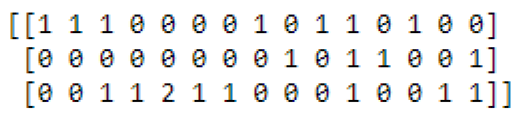
```
#Import required libraries :
from sklearn.feature_extraction.text import CountVectorizer

#Text for analysis :
sentences = ["Jim and Pam travelled by the bus:",
             "The train was late",
             "The flight was full.Travelling by flight is expensive"]

#Create an object :
cv = CountVectorizer()

#Generating output for Bag of Words :
B_O_W = cv.fit_transform(sentences).toarray()

#Total words with their index in model :
print(cv.vocabulary_)
print("\n")

#Features :
print(cv.get_feature_names())
print("\n")

#Show the output :
print(B_O_W)
```

Figure 110: Python implementation code snippet of our bag of words.



```
{'jim': 7, 'and': 0, 'pam': 9, 'travelled': 12, 'by': 2, 'the': 10, 'bus': 1, 'train': 11, 'was': 14, 'late': 8, 'flight': 4,
'full': 5, 'travelling': 13, 'is': 6, 'expensive': 3}


['and', 'bus', 'by', 'expensive', 'flight', 'full', 'is', 'jim', 'late', 'pam', 'the', 'train', 'travelled', 'travelling', 'wa
s']


[[1 1 1 0 0 0 0 1 0 1 1 0 1 0 0]
 [0 0 0 0 0 0 0 0 1 0 1 1 0 0 1]
 [0 0 1 1 2 1 1 0 0 0 1 0 0 1 1]]
```

Figure 111: Output of our bag of words.



```
[[1 1 1 0 0 0 0 1 0 1 1 0 1 0 0]
 [0 0 0 0 0 0 0 0 1 0 1 1 0 0 1]
 [0 0 1 1 2 1 1 0 0 0 1 0 0 1 1]]
```

Figure 112: Output of our bag of words.

Applications:

Natural language processing.

Information retrieval from documents.

Classifications of documents.

Limitations:

Semantic meaning: It does not consider the semantic meaning of a word. It ignores the context in which the word is used.

Vector size: For large documents, the vector size increase, which may result in higher computational time.

Preprocessing: In preprocessing, we need to perform data cleansing before using it.