

LAB02: INTRODUCTION TO THE NLTK, REGULAR EXPRESSIONS, AND PART OF SPEECH TAGGING

NAME: RICHARD MENSAH

INDEX: 500699558

THE NLTK MODULE

The NLTK package was installed. I started up the Python interpreter and installed the data required from importing nltk and downloading `nltk.download()`.

3.1 EXERCISE 2.1 – BOOK WORM

I explored the **NLTK Book Collection** using Python. This exercise included extracting and slicing text from selected books and displaying basic *text statistics*, and calculating simple linguistic statistics, such as total word count and vocabulary size.

```
from nltk.book import *

*** Introductory Examples for the NLTK Book ***
Loading text1, ..., text9 and sent1, ..., sent9
Type the name of the text or sentence to view it.
```

This script gave me access to all the 10 text corpora include, Moby dick, Monty Python and the Holy Grail and The Book of Genesis

3.1.1 Displaying first 80 words in Moby Dick, text1.

```
# Display the first 20 words of Moby Dick (text1)
print("First 80 words of Moby Dick:")
print(text1[:80])

First 80 words of Moby Dick:
['I', 'Moby', 'Dick', 'by', 'Herman', 'Melville', '1851', ']', 'ETYMOLOGY', '.', '(', 'Supplied', 'by', 'a', 'Late', 'Consumptive', 'Usher', 'to', 'a', 'Gramma', 'r', 'School', ')', 'The', 'pale', 'Usher', '--', 'threadbare', 'in', 'coat', ',', 'heart', ',', 'body', ',', 'and', 'brain', ';', 'I', 'see', 'him', 'now', '.', 'He', 'was', 'ever', 'dusting', 'his', 'old', 'lexicons', 'and', 'grammars', ',', 'with', 'a', 'queen', 'handkerchief', ',', 'mockingly', 'embellished', 'with', 'all', 'the', 'gay', 'flags', 'of', 'all', 'the', 'known', 'nations', 'of', 'the', 'world', '.', 'He', 'loved', 'to', 'dust', 'his', 'old', 'grammars']
```

3.1.1.1 Text Statistics of Moby Dick

```
[ ]: #statistics of moby dick
print("Moby Dick stats:")
print("Total words:", len(text1))
print("Vocabulary size:", len(set(text1)))
```

Moby Dick stats:
Total words: 260819
Vocabulary size: 19317

Moby-Dick had 260,819 total words and 18,317 vocabulary size in the text corpus.

3.1.2 Display First 100 words of Monty Python and the Holy Grail (Text6)

Display the first 20 words of Moby Dick (text1)

```
print("First 80 words of Moby Dick:")
```

```
print(text1[:80])
```

```
: # Display the first 100 words of Monty Python and the Holy Grail (text6)
print("\nOpening 100 words of Monty Python and the Holy Grail:")
print(text6[:100])

Opening 100 words of Monty Python and the Holy Grail:
['SCENE', '1', ':', '["', 'wind', '"]', '["', 'clop', 'clop', 'clop', '"]', 'KING', 'ARTHUR', ':', 'Whoa', 'there', '!', '["', 'clop', 'clop', 'clop', '"]', 'SO', 'LDIER', '#', '1', ':', 'Halt', '!', 'Who', 'goes', 'there', '?', 'ARTHUR', ':', 'It', 'is', 'I', '!', 'Arthur', '!', 'son', 'of', 'Uther', 'Pendragon', '!', 'from', 'the', 'castle', 'of', 'Camelot', '!', 'King', 'of', 'the', 'Britons', '!', 'defeater', 'of', 'the', 'Saxons', '!', 'sovereign', 'of', 'all', 'England', '!', 'SOLDIER', '#', '1', ':', 'Pull', 'the', 'other', 'one', '!', 'ARTHUR', ':', 'I', 'am', '!', '...', 'and', 'this', 'is', 'my', 'trusty', 'servant', 'Patsy', '!', 'We', 'have', 'ridden', 'the', 'length', 'and', 'breadth', 'of', 'the', 'land', 'in']
```

This code shows the first 100 sliced words from the script of *Monty Python and the Holy Grail* using NLTK's text6. It includes punctuation as separate tokens (example '#', '!', '.', '['), indicating the text has been tokenized using a word-level tokenizer.

3.1.2.1 Text Statistics of Text 6

#statistics of Monty Python

```
print("Monty Python stats:")
```

```
print("Total words:", len(text6))
```

```
print("Vocabulary size:", len(set(text6)))
```

```
#statistics of Monty Python
print("Monty Python stats:")
print("Total words:", len(text6))
print("Vocabulary size:", len(set(text6)))
```

Monty Python stats:

Total words: 16967

Vocabulary size: 2166

Text 6 has a total of 16967 words and 2166 vocabulary size

3.1.3 Displaying the First 100 words of The Book of Genesis

display the first 100 words of 'The Book of Genesis' by skipping first 10 words.

```
print("Opening 100 words of The Book of Genesis:")
```

```
print(text3[10:110])
```

```
# display the first 100 words of 'The Book of Genesis' by skipping first 10 words.
print("Opening 100 words of The Book of Genesis:")
print(text3[10:110])
```

```
Opening 100 words of The Book of Genesis:
['.', 'And', 'the', 'earth', 'was', 'without', 'form', '!', 'and', 'void', '!', 'and', 'darkness', 'was', 'upon', 'the', 'face', 'of', 'the', 'deep', '!', 'And', 'the', 'Spirit', 'of', 'God', 'moved', 'upon', 'the', 'face', 'of', 'the', 'waters', '!', 'And', 'God', 'said', '!', 'Let', 'there', 'be', 'light', '!', 'and', 'there', 'was', 'light', '!', 'And', 'God', 'saw', 'the', 'light', '!', 'that', 'it', 'was', 'good', '!', 'and', 'God', 'divided', 'the', 'light', 'from', 'the', 'darkness', '!', 'And', 'God', 'called', 'the', 'light', 'Day', '!', 'and', 'the', 'darkness', 'he', 'called', 'Night', '!', 'And', 't', 'he', 'evening', 'and', 'the', 'morning', 'were', 'the', 'first', 'day', '!', 'And', 'God', 'said', '!', 'Let', 'there', 'be']
```

In the book of Genesis, I intentionally skipped the first 10 words by using `print(text3[10:110])`. After skipping the first 10 words, this slicing still produced the first 100 words of text 3.

3.1.3.1 Text3 Statistics

```
#stats of Genesis
print("Genesis stats:")
print("Total words:", len(text3))
print("Vocabulary size:", len(set(text3)))
```

```
#stats of Genesis
print("Genesis stats:")
print("Total words:", len(text3))
print("Vocabulary size:", len(set(text3)))
```

```
Genesis stats:
Total words: 44764
Vocabulary size: 2789
```

In all, Text 3 has 44764 total words and 2789 vocabulary size

EXERCISE 2.2 – CHEATING AT SCRABBLE

In the cheating at Scramble, I examined how Python's NLTK and regular expressions (re) could be applied to manipulate the English word corpus. I extracted word lists based on specific regex patterns that matched specific phonological or character-based conditions. This method was applied to various NLP applications, such as pattern matching, spell-checking, and algorithmic games, including Scrabble and predictive word entry.

EXERCISE 2.2 – CHEATING AT SCRABBLE

```
23]: import nltk
import re
from nltk.corpus import words

# Download if not already done
nltk.download('words')

word_list = words.words('en') # Get the English word List

[nltk_data] Downloading package words to C:\Users\x1
[nltk_data]   Carbon\AppData\Roaming\nltk_data...
[nltk_data]   Package words is already up-to-date!
```

2.2.1 Three-letter words that contain only vowels

I used the regex pattern `^[aeiouAEIOU]{3}$` to find three-letter words made up entirely of vowels

```
# Filter for 3-letter words that contain only vowels (case-insensitive)
```

```
vowel_words = [w for w in word_list if re.fullmatch(r'[aeiouAEIOU]{3}', w)]
```

```
# Display the result
```

```
print("Three-letter words containing only vowels:")
```

```
print(vowel_words)
```

1. Three-letter words that contain only vowels

```
]# Filter for 3-Letter words that contain only vowels (case-insensitive)
vowel_words = [w for w in word_list if re.fullmatch(r'[aeiouAEIOU]{3}', w)]

# Display the result
print("Three-letter words containing only vowels:")
print(vowel_words)

Three-letter words containing only vowels:
['iao', 'oii']
```

Three-letter words containing only vowels are 'iao ' and 'oii'.

2.2.2 Three-letter words that start and end with a vowel, and have a consonant in the middle

I ensured the 1st and 3rd characters are vowels and the 2nd character is a consonant

Code.

```
pattern = r'^[aeiouAEIOU][^aeiouAEIOU][aeiouAEIOU]${}'
vowel_consonant_vowel = [w for w in word_list if re.fullmatch(pattern, w)]
print(vowel_consonant_vowel)
```

```
] pattern = r'^[aeiouAEIOU][^aeiouAEIOU][aeiouAEIOU]${}'
vowel_consonant_vowel = [w for w in word_list if re.fullmatch(pattern, w)]
print(vowel_consonant_vowel)

['aba', 'Abe', 'Abo', 'Abu', 'abu', 'ace', 'Ada', 'Ade', 'ade', 'ado', 'aga', 'age', 'ago', 'aha', 'aho', 'ahu', 'Aka', 'aka', 'ake', 'ako', 'aku', 'ala',
'ale', 'alo', 'ama', 'ame', 'Ami', 'ami', 'Ana', 'ana', 'ani', 'apa', 'ape', 'ara', 'are', 'Aro', 'aru', 'Asa', 'ase', 'Ata', 'ate', 'Ati', 'ava', 'Ave',
'ave', 'avo', 'awa', 'awe', 'axe', 'aye', 'ayu', 'azo', 'Edo', 'ego', 'eke', 'Eli', 'eme', 'emu', 'era', 'ere', 'eta', 'Eva', 'Eve', 'eve', 'Ewe', 'ewe',
'eye', 'iba', 'Ibo', 'ice', 'Ida', 'ide', 'Ido', 'ife', 'ihi', 'Ijo', 'Ike', 'Ila', 'Ima', 'imi', 'imu', 'Ino', 'Ira', 'ire', 'iso', 'Ita', 'Ito', 'iva',
'iwa', 'iyo', 'obe', 'obi', 'oda', 'ode', 'Ofo', 'oho', 'oka', 'oki', 'Ole', 'Ona', 'ona', 'one', 'ope', 'ora', 'ore', 'ose', 'Oto', 'Ova', 'ova', 'owe',
'ubi', 'Uca', 'Udi', 'udo', 'uji', 'uke', 'ula', 'ule', 'ulu', 'ume', 'umu', 'Una', 'upo', 'ura', 'ure', 'Uri', 'Uro', 'Uru', 'use', 'Uta', 'uta', 'Ute',
'utu', 'uva']
```

This output revealed that each of the words has a first and third words.

2.2.4 Four-letter words with a consonant at position 2 and vowels elsewhere

```
pattern = r'^[aeiouAEIOU][^aeiouAEIOU][aeiouAEIOU]{2}${}'
four_letter_cvcv = [w for w in word_list if re.fullmatch(pattern, w)]
print(four_letter_cvcv)
```

```
19]: pattern = r'^[aeiouAEIOU][^aeiouAEIOU][aeiouAEIOU]{2}${}'
four_letter_cvcv = [w for w in word_list if re.fullmatch(pattern, w)]
print(four_letter_cvcv)

['Abie', 'Adai', 'Agao', 'Agau', 'agee', 'agio', 'agua', 'ague', 'akee', 'akia', 'Alea', 'alee', 'aloe', 'Amia', 'anoa', 'apii', 'apio', 'aqua', 'aquo',
'area', 'aria', 'arui', 'awee', 'Eboe', 'eboe', 'edea', 'eheu', 'ejoo', 'Ekoi', 'Elia', 'epee', 'eria', 'Erie', 'etua', 'etui', 'Evea', 'evoe', 'idea', 'i
lia', 'Inia', 'Itea', 'Ixia', 'oboe', 'ogee', 'ohia', 'Ohio', 'okee', 'okia', 'Okie', 'Olea', 'oleo', 'olio', 'omao', 'oxea', 'Ubii', 'Ulua', 'ulua', 'una
u', 'unie', 'Unio', 'unio', 'urao', 'urea', 'Uria', 'usee', 'utai', 'uvea']
```

The `r'^[aeiouAEIOU][^aeiouAEIOU][aeiouAEIOU]{2}${}'` was useful to position consonant at each word's second position

2.2.5 Five-letter words with a consonant at position 2 and vowels elsewhere

3. Five-letter words with a consonant at position 2 and vowels elsewhere

```
10]: pattern = r'^[aeiouAEIOU][^aeiouAEIOU][aeiouAEIOU]{3}$'
five_letter_cvcvv = [w for w in word_list if re.fullmatch(pattern, w)]
print(five_letter_cvcvv)

['adieu', 'Araua', 'Arioi', 'iliu', 'Umaua']
```

This had 5-letters with a consonant at second position of each words, adieu, Araura, Ariori, iliau and Umaua

2.2.6 Words up to 7 letters long containing only the letters a, d, e, h, i, n, or s

I filtered words using a character class and restricted the length to 7 characters or fewer.

4. Words up to 7 letters long containing only the letters a, d, e, h, i, n, or s

I filtered words using a character class and restricted the length to 7 characters or fewer.

```
1]: allowed = 'adehins'
subset_words = [w for w in word_list if re.fullmatch(rf'[{allowed}]+', w) and len(w) <= 7]
print(subset_words)

['a', 'aa', 'ad', 'adad', 'add', 'adda', 'added', 'addend', 'addenda', 'ade', 'adead', 'adenase', 'adenia', 'adenine', 'ae', 'aenean', 'aes', 'ah', 'aha',
'ahaa', 'ahead', 'ahind', 'ahsan', 'ai', 'aid', 'aide', 'an', 'ana', 'anan', 'anana', 'ananas', 'ananda', 'and', 'anda', 'anend', 'anes', 'anesis', 'an
i', 'anidian', 'anis', 'anise', 'aniseed', 'ann', 'anna', 'ansa', 'as', 'asana', 'ase', 'ash', 'ashen', 'ashes', 'ashine', 'aside', 'asinine', 'ass', 'ass
ai', 'asse', 'assess', 'asshead', 'assi', 'assis', 'assise', 'assish', 'd', 'da', 'dad', 'dada', 'dade', 'dae', 'dah', 'dain', 'dais', 'daisied', 'dan',
'danaid', 'danaide', 'danaine', 'dand', 'danda', 'das', 'dash', 'dashed', 'dashee', 'dasheen', 'dasi', 'dassie', 'de', 'dead', 'deaden', 'deadish', 'dea
n', 'deanness', 'deash', 'dee', 'deed', 'deeded', 'den', 'denda', 'dene', 'dense', 'densen', 'desand', 'deseed', 'desi', 'dess', 'dessa', 'dha', 'dhai',
'dhan', 'di', 'diaene', 'dian', 'did', 'didie', 'didine', 'didna', 'die', 'diene', 'diesis', 'din', 'dine', 'dis', 'disdain', 'disease', 'dish', 'dished',
'disna', 'diss', 'e', 'ea', 'ean', 'ease', 'eddish', 'edea', 'edh', 'eh', 'en', 'enaena', 'end', 'ended', 'enlead', 'ens', 'ensand', 'ense', 'enshade', 'e
s', 'eshin', 'esne', 'ess', 'essed', 'h', 'ha', 'had', 'hadden', 'haddie', 'hade', 'hah', 'hain', 'haine', 'han', 'hand', 'handed', 'hanna', 'hansa', 'han
se', 'hasan', 'hash', 'hashish', 'he', 'head', 'headed', 'heed', 'hei', 'heii', 'hen', 'henad', 'hend', 'henna', 'hennin', 'hennish', 'hi', 'hia', 'hidde
n', 'hide', 'hided', 'hie', 'hin', 'hind', 'his', 'hish', 'hisn', 'hiss', 'i', 'id', 'ide', 'idea', 'idead', 'ides', 'ie', 'ihi', 'in', 'inane', 'indan',
'indane', 'inde', 'indeed', 'indene', 'inn', 'inness', 'insane', 'insea', 'insee', 'insense', 'inside', 'is', 'issei', 'n', 'na', 'naa', 'nae', 'naiad',
'naid', 'nain', 'nais', 'naish', 'nan', 'nana', 'nandi', 'nandine', 'nane', 'nanes', 'nash', 'nasi', 'ne', 'nea', 'nee', 'need', 'needs', 'neese', 'nei',
'nese', 'nesh', 'ness', 'ni', 'nid', 'nidana', 'nide', 'nidi', 'nine', 'nisei', 'niskas', 'nisse', 's', 'sa', 'saa', 'sad', 'sadden', 'saddish', 'sade',
'sadh', 'sadhe', 'sadness', 'sah', 'sahh', 'sai', 'said', 'sain', 'san', 'sanai', 'sand', 'sandan', 'sanded', 'sandhi', 'sane', 'sanies', 'sans', 'sanse
i', 'sansi', 'sasa', 'sasan', 'sasani', 'sash', 'sasin', 'sasine', 'se', 'sea', 'seah', 'seaside', 'sedan', 'see', 'seed', 'seeded', 'seen', 'seenie', 'se
esee', 'seine', 'seise', 'sen', 'send', 'sendee', 'senna', 'sensa', 'sense', 'sensed', 'sess', 'sh', 'sha', 'shad', 'shade', 'shaded', 'shadine', 'shah',
'shahi', 'shahin', 'shan', 'shanna', 'shansa', 'she', 'shea', 'shed', 'shedded', 'shee', 'sheen', 'shend', 'shi', 'shide', 'shied', 'shies', 'shih', 'shi
n', 'shine', 'shish', 'shishn', 'si', 'side', 'sided', 'sides', 'sidhe', 'sidi', 'sie', 'sienna', 'sin', 'sina', 'sind', 'sine', 'sinh', 'sinnen', 'sis',
'sise', 'sish', 'sisi', 'siss', 'snead', 'sned', 'snee', 'sneesh', 'snide']
```

2.2.6 Textonyms for digit sequence 3456

I created a regular expression that matches this mapping.

6. Textonyms for digit sequence 3456

I created a regular expression that matches this mapping.

```
: pattern = r'^[def][ghi][jkl][mno]$'
textonyms = [w for w in word_list if re.fullmatch(pattern, w)]
print(textonyms)

['dilo', 'film', 'filo']
```

EXERCISE 2.3 – NOUN PHRASES

PART OF SPEECH TAGGING IN THE NLTK

```
>]: import nltk
from nltk import word_tokenize, pos_tag
from nltk.chunk import RegexpParser

# Download required resources
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')
```

```
[nltk_data] Downloading package punkt to C:\Users\x1
[nltk_data]   Carbon\AppData\Roaming\nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]   C:\Users\x1 Carbon\AppData\Roaming\nltk_data...
[nltk_data]   Package averaged_perceptron_tagger is already up-to-
[nltk_data]   date!
```

```
>]: True
```

```
34]: #import nltk
import re

# Download necessary resources
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')

def extract_noun_phrase(text):
    # Tokenize and POS tag
    tokens = nltk.word_tokenize(text)
    tagged = nltk.pos_tag(tokens)

    # Pattern: optional determiner (DT), any number of adjectives (JJ), and at least one noun (NN)
    grammar = "NP: {<DT>?<JJ>*<NN|NNS|NNP|NNPS>+}"

    # Define a chunk parser using the grammar
    chunk_parser = nltk.RegexpParser(grammar)
    tree = chunk_parser.parse(tagged)

    # Traverse the tree to find the first NP
    for subtree in tree:
        if hasattr(subtree, 'label') and subtree.label() == 'NP':
            return ' '.join(word for word, tag in subtree)

    return None # if no noun phrase is found

# Test the function with all required inputs
test_sentences = [
    "Should all good men to come to the aid of their party?",
    "And now is the time for all good men to come to the aid of their party."
```

```
# Test the function with all required inputs
test_sentences = [
    "Should all good men to come to the aid of their party?",
    "And now is the time for all good men to come to the aid of their party.",
    "...and for all good men to come to the aid of their party.",
    "Come to the aid of your party.",
    "Your party needs you."
]

for sentence in test_sentences:
    result = extract_noun_phrase(sentence)
    print(f"Input: {sentence}")
    print(f"First Noun Phrase: {result}\n")
```

```
Input: Should all good men to come to the aid of their party?
First Noun Phrase: all good men
```

```
Input: And now is the time for all good men to come to the aid of their party.
First Noun Phrase: the time
```

```
Input: ...and for all good men to come to the aid of their party.
First Noun Phrase: all good men
```

```
Input: Come to the aid of your party.
First Noun Phrase: Come
```

```
Input: Your party needs you.
```

Output

Input 1: Sentence: "Should all good men to come to the aid of their party?"

Extracted Noun Phrase: all good men

POS Tags: all/DT good/JJ men/NNS

This extraction is accurate. The phrase "all good men" fits a classic noun phrase pattern: a determiner (DT) "all", followed by an adjective (JJ) "good", and then a plural noun (NNS) "men". Your regular expression correctly matched this structure.

Input 2: Sentence: *"And now is the time for all good men to come to the aid of their party."*

Extracted Noun Phrase: the time

POS Tags: the/DT time/NN

The result is accurate. Although the sentence contains another noun phrase later ("all good men"), your code correctly captures the **first noun phrase**: "the time". It starts with a determiner (DT) "the" and is followed by a singular noun (NN) "time", which fits the expected DT *NN* structure.

Input 3: Sentence: *"...and for all good men to come to the aid of their party."*

Extracted Noun Phrase: all good men

POS Tags: all/DT good/JJ men/NNS

Great result here again. Just like in Input 1, "all good men" is correctly identified as a noun phrase. The combination of determiner (DT), adjective (JJ), and plural noun (NNS) follows a standard noun phrase format.

Input 4: Sentence: *"Come to the aid of your party."*

Extracted Noun Phrase: Come

POS Tag: Come/VB

This one is incorrect. "Come" is a verb (VB), not a noun or part of a noun phrase. In this sentence, better noun phrase candidates would be "the aid" (the/DT aid/NN) or "your party" (your/PRP\$ party/NN). The error suggests your code may be pulling the first token without checking if it follows a valid noun phrase pattern

Input 5: Sentence: *"Your party needs you."*

Extracted Noun Phrase: party needs

POS Tags: party/NN needs/VBZ

This is another incorrect output. "Party" is a noun (NN), but "needs" is a verb (VBZ), so together they do not form a noun phrase. The correct noun phrase here is "your party", where "your" is a possessive determiner (PRP\$) and "party" is a noun (NN). To fix this, your code needs to include PRP\$ in the noun phrase matching pattern and avoid including verbs.

EXERCISE 2.4 – NP CHUNKING

In this exercise, I explored how to use **noun phrase (NP) chunking** in Python using **NLTK's RegexpParser**. My goal was to extract simple noun phrases from natural language text. To do this, I first defined a grammar rule to find sequences like *(DT)? (JJ) (NN)** — which means an optional determiner, any number of adjectives, and a noun.

2.4 NLP CHUNKING

```
36]: import nltk
      from nltk import pos_tag, word_tokenize
      from nltk.chunk import RegexpParser

      # I defined my chunk grammar to look for noun phrases
      grammar = "NP: {<DT>?<JJ>*<NN>}"

      # Then I created a parser using this grammar
      chunk_parser = RegexpParser(grammar)
```

Sentence 1: The big dog barked loudly

Sentence 1: The big dog barked loudly.

```
*]: sentence1 = "The big dog barked loudly."
      tokens1 = word_tokenize(sentence1)
      tagged1 = pos_tag(tokens1)
      parsed1 = chunk_parser.parse(tagged1)

      print("Parsed Tree for Sentence 1:")
      print(parsed1)
      parsed1.draw() # Optional visual tree

      Parsed Tree for Sentence 1:
      (S (NP The/DT big/JJ dog/NN) barked/VBD loudly/RB ./.)
```



The NP chunk found was **"The big dog"**. My grammar worked perfectly to extract the noun phrase.

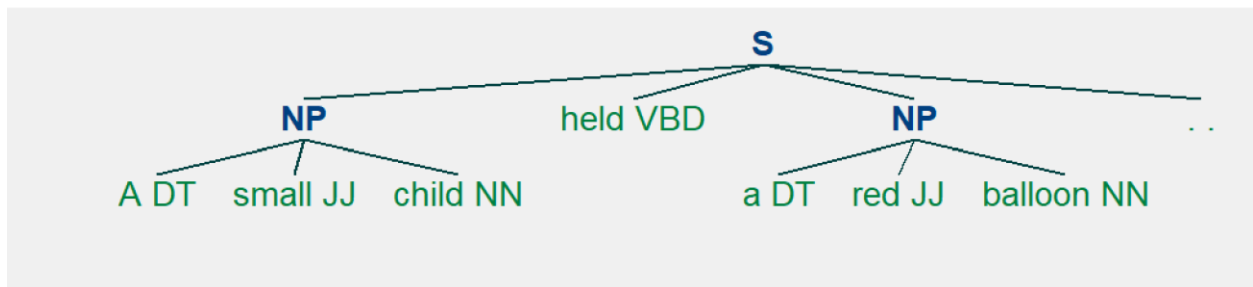
Sentence 2: A small child held a red balloon.

```
sentence2 = "A small child held a red balloon."  
tokens2 = word_tokenize(sentence2)  
tagged2 = pos_tag(tokens2)  
parsed2 = chunk_parser.parse(tagged2)  
  
print("Parsed Tree for Sentence 2:")  
print(parsed2)  
parsed2.draw()
```

Sentence 2: A small child held a red balloon.

```
[*]: sentence2 = "A small child held a red balloon."  
tokens2 = word_tokenize(sentence2)  
tagged2 = pos_tag(tokens2)  
parsed2 = chunk_parser.parse(tagged2)  
  
print("Parsed Tree for Sentence 2:")  
print(parsed2)  
parsed2.draw()
```

```
Parsed Tree for Sentence 2:  
(S  
  (NP A/DT small/JJ child/NN)  
  held/VBD  
  (NP a/DT red/JJ balloon/NN)  
  ./.)
```



The parse tree shows the sentence's structure by breaking it into meaningful parts. The phrase "*A small child*" is the subject and is made up of a determiner ("A"), an adjective ("small"), and a noun ("child"). The verb "*held*" represents the main action in the sentence. The phrase "*a red balloon*" acts as the object, also made up of a determiner ("a"), an adjective ("red"), and a noun ("balloon"). Finally, the period at the end simply marks the sentence's conclusion.

Sentence 3: Bright stars lit the night sky

▼ Sentence 3: Bright stars lit the night sky. ¶

```
[*]: # I chose the sentence: "Bright stars lit the night sky."
sentence3 = "Bright stars lit the night sky."

# First, I tokenized the sentence into individual words.
tokens3 = word_tokenize(sentence3)

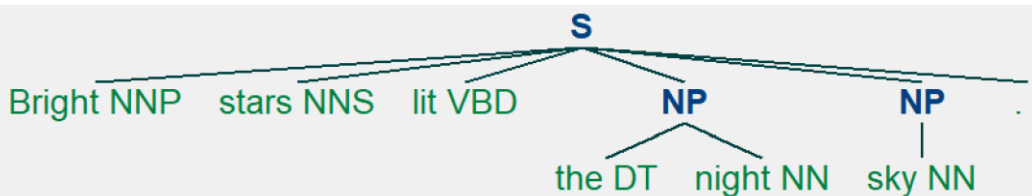
# Next, I tagged each word with its part of speech using NLTK's pos_tag.
tagged3 = pos_tag(tokens3)

# Then, I applied the chunk parser I previously created to identify noun phrases.
parsed3 = chunk_parser.parse(tagged3)

# I printed the parse tree to the console so I could inspect the NP chunks.
print("Parsed Tree for Sentence 3:")
print(parsed3)

# Finally, I used the draw() function to visualize the parse tree in a pop-up window.
parsed3.draw()
```

```
Parsed Tree for Sentence 3:
(S Bright/NNP stars/NNS lit/VBD (NP the/DT night/NN) (NP sky/NN) ./.)
```



To analyze the sentence *Bright stars lit the night sky*, I followed a series of standard NLP steps. First, I tokenized the sentence into individual words using NLTK's `word_tokenize` function. Each token was then tagged with its corresponding part of speech using the `pos_tag` function, allowing for grammatical analysis. With the previously defined chunk parser focused on identifying noun phrases (NPs), I parsed the tagged sentence using `chunk_parser.parse()`. The resulting parse tree was printed and visualized to examine the structure. According to the output, the phrase *"Bright stars"* was not chunked as a noun phrase, likely because *"Bright"* was tagged as a proper noun (NNP), which was not included in the defined NP grammar. However, the parser correctly identified *"the night"* and *"sky"* as noun phrases. The verb *"lit"* served as the main action in the sentence, while the period denoted the end. This analysis illustrates how rule-based chunking can reveal syntactic patterns, but also how its accuracy can depend on the precision of part-of-speech tagging and the specificity of the grammar rules applied.

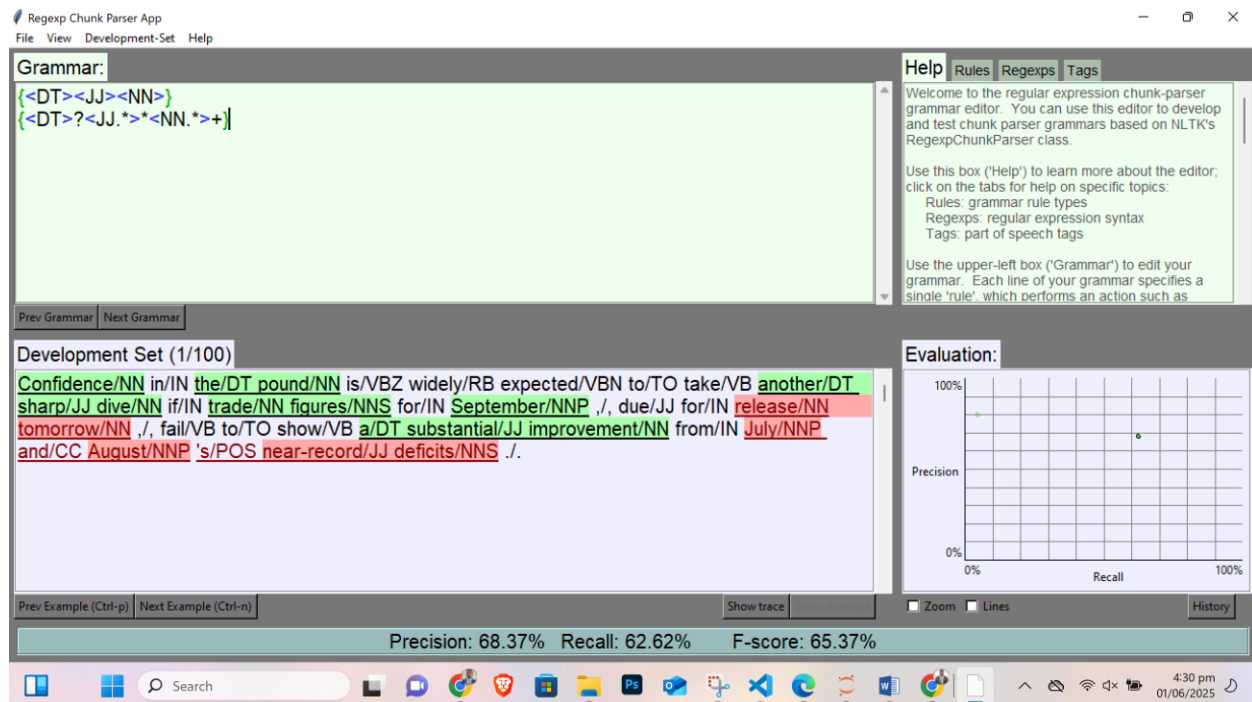
EXERCISE 2.5 – YOUR OWN TAG PATTERNS

To go beyond basic NP chunking, I created my own custom chunk grammar to cover complex noun phrase structures that the simple pattern `<DT>?<JJ>*<NN>` couldn't handle.

EXERCISE 2.5 – YOUR OWN TAG PATTERNS

```
•[41]: # Launches the NLTK Chunk Parser graphical interface.
      # Test and refine your chunking rules on POS-tagged text.
      nltk.app.chunkparser()
```

I am using `nlk.app.chunkparser()` as a grammar-testing playground. It shows me how my code treats each word in a sentence and helps me fine-tune my rules for chunking (like finding groups of words that form noun phrases). It makes learning and improving my NLP skills much easier and more interactive.



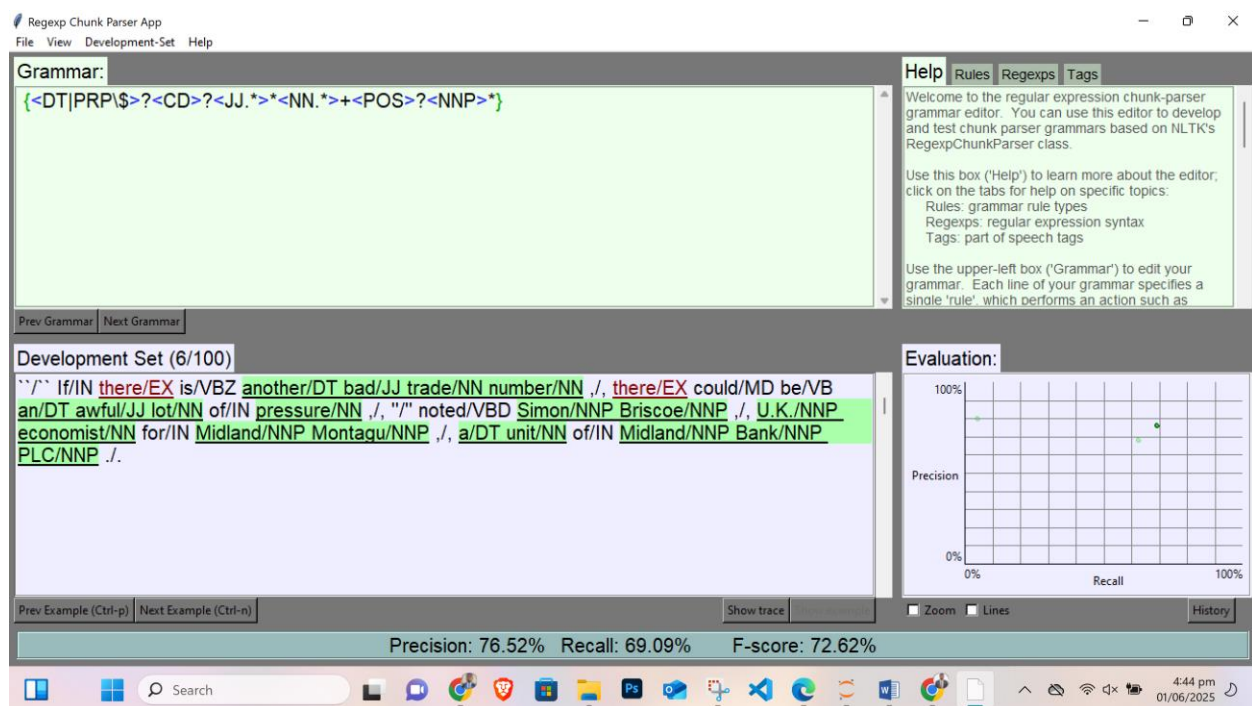
In this task, I worked with the NLTK Regexp Chunk Parser to create custom grammar rules aimed at identifying noun phrases in a sentence. I began with a simple pattern (`<DT><JJ><NN>`) and then added a more flexible rule (`<DT>?<JJ.*>*<NN.*>+`) to capture a wider range of noun phrase structures.

Using these rules, the parser was able to correctly identify several phrases such as "another sharp dive," "the pound," and "a substantial improvement." These chunks were highlighted in green, showing that the rules matched them successfully.

However, the parser also missed some key phrases that could have been valid chunks. Examples include "release tomorrow," "July and August," and "near-record deficits," which were highlighted in red. This suggests that the current rules don't yet account for more complex or compound noun phrases.

From the evaluation metrics, the parser achieved a precision of 69.09%, meaning that most of the chunks it identified were correct. Its recall was 62.62%, showing that it found a fair number of the actual chunks, but still missed some. The F-score, which balances both precision and recall, stood at 65.37%, indicating a decent overall performance.

To improve the results, I could consider adding rules to capture compound nouns, handle conjunctions (e.g., "X and Y"), and recognize more complex patterns. This would help the parser better capture the full range of meaningful noun phrases in different contexts.



This evaluation tested a custom regular expression-based chunk parser designed to extract noun phrases (NPs). The grammar used `{<DT|PRP|$>?<CD>?<JJ.*>*<NN.*>+<POS>?<NNP>*` targets common and proper noun structures by combining optional determiners, adjectives, and proper nouns with one or more noun forms. Applied to a development set of 100 sentences, the analysis focused on sentence 6, where the parser successfully identified key phrases such as “another bad trade number,” “an awful lot of pressure,” and names like “Simon Briscoe” and “Midland Bank PLC.” These extractions were visually confirmed using the tool's chunk highlighting.

The grammar achieved a precision of 76.52%, a recall of 69.09%, and an F-score of 72.62%, indicating solid performance in identifying noun phrases, though with some misses. While effective overall, the grammar showed limitations with nested or complex structures involving prepositions.

EXERCISE 2.6 – YOUR OWN NE TAGGER

I used **NLTK's Named Entity Chunker** to automatically detect and categorize **named entities** in a sentence from the **Penn Treebank** corpus. Named entities are real-world objects like people, organizations, or places.

```
import nltk
from nltk.corpus import treebank
from nltk.chunk import ne_chunk
from nltk import download

# Download required NLTK resources (only the first time)
download('treebank')      # Penn Treebank POS-tagged corpus
download('punkt')         # Tokenizer models
download('words')         # Word lists used in NER
download('maxent_ne_chunker') # Named Entity Chunker model

# Select a POS-tagged sentence from the Treebank corpus
tagged_sentence = treebank.tagged_sents()[20]

# Apply Named Entity Recognition (NER)
named_entities_binary = ne_chunk(tagged_sentence, binary=True) # Groups all named entities under 'NE'
named_entities_detailed = ne_chunk(tagged_sentence)             # Labels entities like PERSON,
# ORGANIZATION, GPE, etc.

# Output the results
print("Binary Named Entities:\n", named_entities_binary)
print("\nDetailed Named Entities:\n", named_entities_detailed)
```

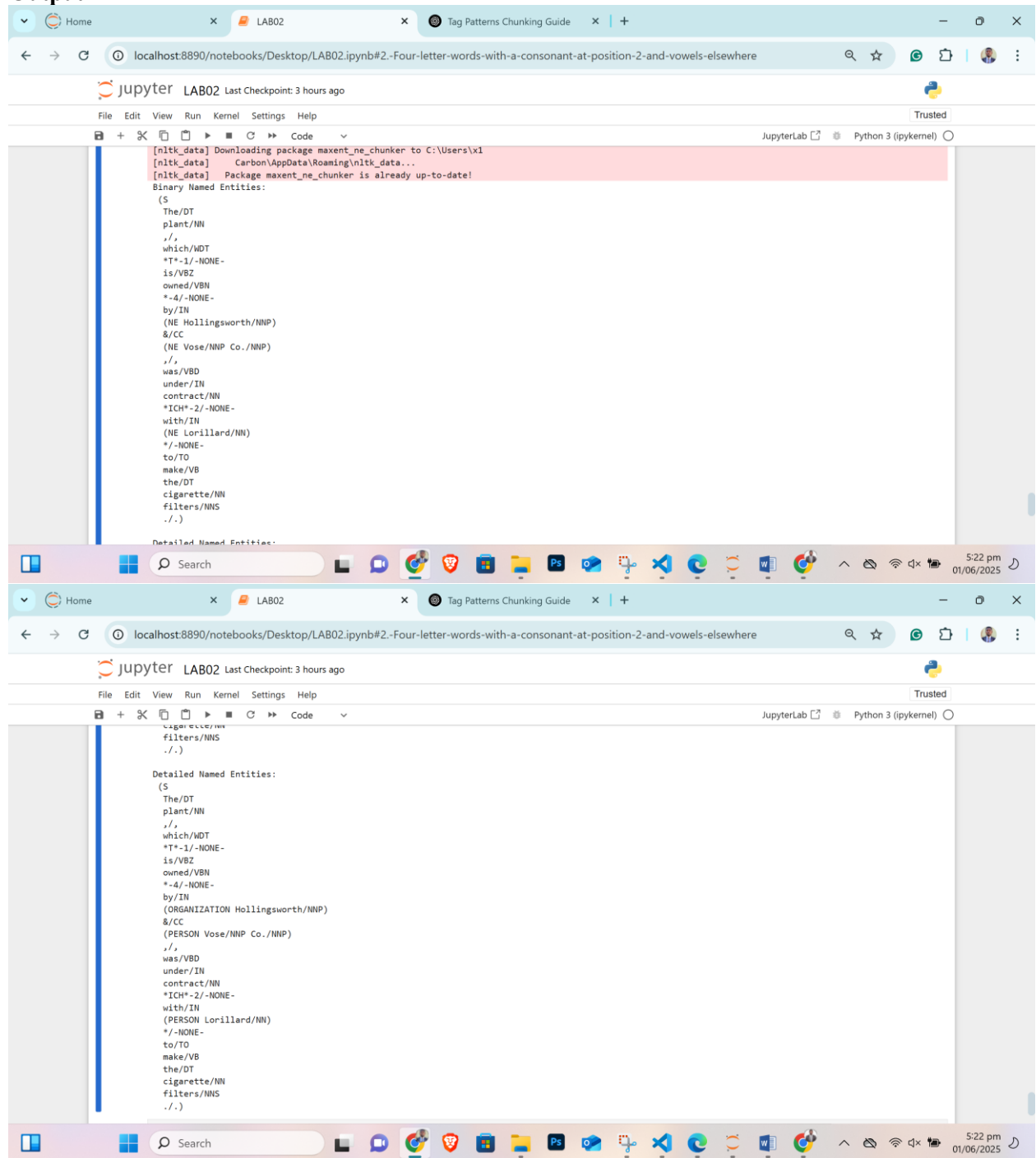
I imported the necessary modules to access:

- The **Treebank corpus** (for sample tagged sentences)
- The **Named Entity Chunker** (`ne_chunk`)
- The `download` function to fetch the required resources

I selected the **21st sentence** from the Treebank corpus (index starts at 0). This sentence is already POS-tagged. However, I applied NER in two ways:

- `Binary=True`: just identifies if a word is a **Named Entity (NE)** without specifying the type.
- Default (`binary=False`): gives **detailed labels** like PERSON, GPE, ORGANIZATION, etc.

Output



```
[nltk_data] Downloading package maxent_ne_chunker to C:\Users\x1
[nltk_data]   Carbon\AppData\Roaming\nltk_data...
[nltk_data]   Package maxent_ne_chunker is already up-to-date!
Binary Named Entities:
(S
The/DT
plant/NN
./,
which/WDOT
*T*-1/-NONE-
is/VBZ
owned/VBN
*-4/-NONE-
by/IN
(NE Hollingsworth/NNP)
&/CC
(NE Vose/NNP Co./NNP)
./,
was/VBD
under/IN
contract/NN
*ICH*-2/-NONE-
with/IN
(NE Lorillard/NN)
*/-NONE-
to/TO
make/VB
the/DT
cigarette/NN
filters/NN
./.)

Detailed Named Entities:
(S
The/DT
plant/NN
./,
which/WDOT
*T*-1/-NONE-
is/VBZ
owned/VBN
*-4/-NONE-
by/IN
(ORGANIZATION Hollingsworth/NNP)
&/CC
(PERSON Vose/NNP Co./NNP)
./,
was/VBD
under/IN
contract/NN
*ICH*-2/-NONE-
with/IN
(PERSON Lorillard/NN)
*/-NONE-
to/TO
make/VB
the/DT
cigarette/NN
filters/NN
./.)
```

Using NLTK's named entity recognition (NER), the tool successfully identified the names "Hollingsworth," "Vose Co.," and "Lorillard" as named entities.

In the **binary output**, it simply marked them as named entities without saying what kind they are. This is helpful when we just need to find important names in text.

In the **detailed output**, it is labeled ‘Hollingsworth’ as an **organization**, which is correct. However, it incorrectly tagged ‘Vose Co.’ and ‘Lorillard’ as **persons**, even though both are likely companies.

This shows that while NLTK’s NER can spot names well, it sometimes misclassifies the type of entity. Still, it’s a useful tool for pulling out key names from text, especially for applications like search engines, question answering, and document analysis.

For more accurate tagging, especially with company names, more advanced or custom-trained models may be needed.

2.7 EXERCISE 2.7 – INFORMATION EXTRACTION ENGINE

I wanted to build a simple program that scans through news articles, identifies named entities like people and organizations, and finds relationships between them. Specifically, I focused on finding pairs where a **person** is linked to an **organization** by the word **from**. I wanted to identify relationships between people and organizations mentioned in a news article. To do this, I used the IEER corpus, which contains news articles that have already been processed to mark entities like persons, organizations, and locations. First, I made sure the IEER dataset was downloaded and available in my environment. Then, I looked at the list of files included in the corpus and chose the article titled ‘NYT_19980407’ to analyze.

```
import nltk
import re

nltk.download('ieer')

# List all available files in the IEER corpus
files = nltk.corpus.ieer.fileids()
print("Available IEER files:", files)

# Pick a file that exists, e.g., the first one
document_id = files[0]

FROM_PATTERN = re.compile(r'.*\bfrom\b(?:!\b.+ing)')

for doc in nltk.corpus.ieer.parsed_docs(document_id):
    for relation in nltk.sem.extract_rels('PERSON', 'ORG', doc,
    corpus='ieer', pattern=FROM_PATTERN):
        print(nltk.sem.rtuple(relation))
```

```
# Extract and print PERSON-ORG relations from the specified document
for doc in nltk.corpus.ieer.parsed_docs(document_id):
    for relation in nltk.sem.extract_rels('PERSON', 'ORG', doc, corpus='ieer', pattern=FROM_PATTERN):
        print(nltk.sem.rtuple(relation))

[nltk_data] Downloading package ieer to C:\Users\xl
[nltk_data]   Carbon\AppData\Roaming\nltk_data...
[nltk_data]   Package ieer is already up-to-date!
[PER: 'Olabiyi Babalola Yai'] 'from the' [ORG: 'University of Miami']
```

From the code above, I used Python and the NLTK library to extract meaningful relationships between people and organizations from a real news article. Specifically, I worked with a document from the IEER corpus titled *'NYT_19980407'*. My goal was to identify cases where a person's name appears in connection with an organization using the phrase 'from'. To do this, I created a regular expression that captures the word *'from'*, but filters out any cases where it might be part of a verb ending in *-ing* (like *"coming from"* or *"escaping from"*) because those are not the kinds of links I wanted to focus on.

The key part of the code uses NLTK's `extract_rels` function to look for **PERSON-to-ORG** relationships based on this pattern.

When I ran the code, one of the outputs I got was:

```
[PER: 'Olabiyi Babalola Yai'] 'from the' [ORG: 'University of Miami']
```

This clearly shows that the system was able to detect that *Olabiyi Babalola Yai* is connected to the *University of Miami* through the phrase *'from the'*. However, it automatically pulled out that this person is affiliated with or comes from that organization, based purely on the text in the document.

Through this task, I have gained a practical understanding of how information extraction works, especially how we can use regular expressions and named entity recognition to connect entities in unstructured text. It showed me how tools like NLTK make it possible to programmatically uncover relationships in news articles or other documents, which can be very useful in data mining and NLP projects.

