**DSA0314-Natural Language Processing for Programming Professionals**

**PROGRAMS FOR PRACTICAL WEEK**

1. Write program demonstrates how to use regular expressions in Python to match and search for patterns in text.

**PROGRAM:**

import re

text1 = "DSA-0314 Natural Language Processing"

text2 = "DSA-0314 Natural Language"

word = "Natural Language Processing"

pattern = fr'\b{word}\b'

match1 = re.search(pattern, text1)

if match1:

print("found the text")

else:

print("not found in text")

match2 = re.search(pattern, text2)

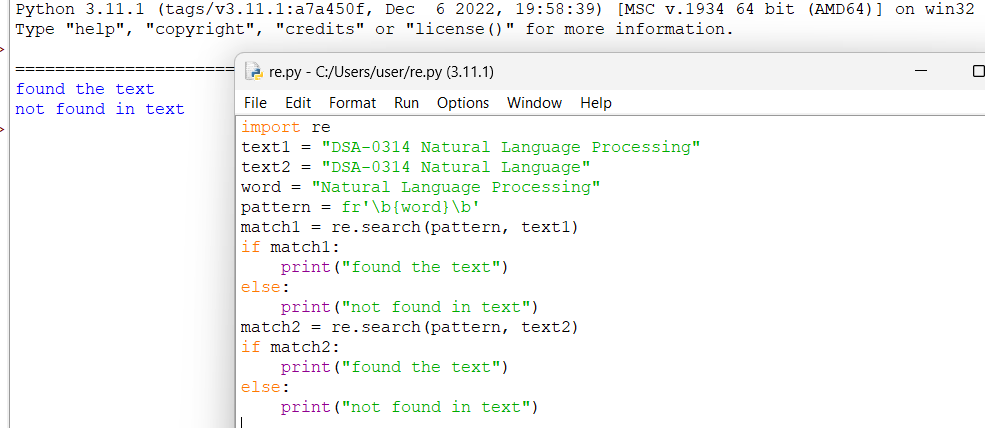
if match2:

print("found the text")

else:

print("not found in text")

**OUTPUT:**



1. Implement a basic finite state automaton that recognizes a specific language or pattern. In this example, we'll create a simple automaton to match strings ending with 'ab' using python.

**PROGRAM:**

INITIAL\_STATE = 'q0'

ACCEPTING\_STATE = 'q2'

TRANSITIONS = {

'q0': {'a': 'q1', 'b': 'q0'},

'q1': {'a': 'q1', 'b': 'q2'},

'q2': {'a': 'q1', 'b': 'q0'}

}

def process\_string(test\_strings):

current\_state = INITIAL\_STATE

for char in test\_strings:

if char in TRANSITIONS.get(current\_state, {}):

current\_state = TRANSITIONS[current\_state][char]

else:

current\_state = INITIAL\_STATE

return current\_state == ACCEPTING\_STATE

test\_strings = ["ab", "aab", "bab", "bbaaab", "a", "b", "abc"]

for string in test\_strings:

if process\_string(string):

print(f"'{string}' is accepted by the FSA.")

else:

print(f"'{string}' is not accepted by the FSA.")

**OUTPUT:**

**A close up of text

Description automatically generated**

1. Write program demonstrates how to perform morphological analysis using the NLTK library in Python.

**PROGRAM:**

import nltk

from nltk.tokenize import word\_tokenize

nltk.download('punkt')

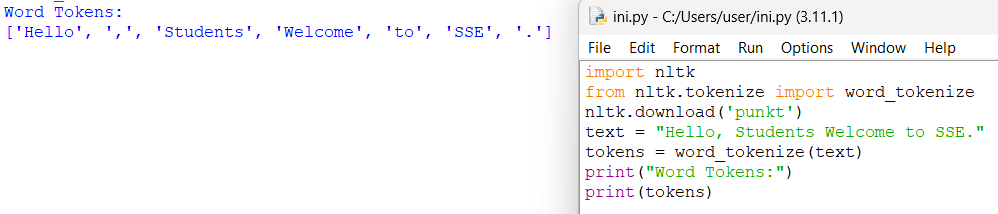
text = "Hello, Students Welcome to SSE."

tokens = word\_tokenize(text)

print("Word Tokens:")

print(tokens)

**OUTPUT:**



1. Implement a finite-state machine for morphological parsing. In this example, we'll create a simple machine to generate plural forms of English nouns using python.

**PROGRAM:**

states = {

"q0": {"other": "q0", "s": "q1"}

}

start\_state = "q0"

final\_states = {"q1"}

def process(word):

current\_state = start\_state

for char in word:

if char == 's':

current\_state = states[current\_state].get('s')

else:

current\_state = states[current\_state].get('other')

return current\_state in final\_states

words = ["cakes", "monkeys", "apple", "banana"]

for word in words:

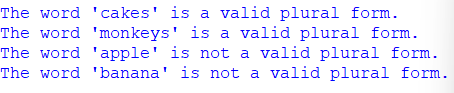
if process(word):

print(f"The word '{word}' is a valid plural form.")

else:

print(f"The word '{word}' is not a valid plural form.")

**OUTPUT:**



1. Use the Porter Stemmer algorithm to perform word stemming on a list of words using python libraries.

**PROGRAM:**

import nltk

from nltk.stem import PorterStemmer

nltk.download('punkt')

stemmer = PorterStemmer()

words = ["running", "jumping", "happiness", "computers", "generous"]

stemmed\_words = [stemmer.stem(word) for word in words]

print(stemmed\_words)

**OUTPUT:**



1. Implement a basic N-gram model for text generation. For example, generate text using a bigram model using python.

**PROGRAM:**

import nltk

nltk.download('treebank')

from nltk.corpus import treebank

from nltk.tag import BigramTagger

train\_sents = treebank.tagged\_sents()[:3000]

bigram\_tagger = BigramTagger(train\_sents)

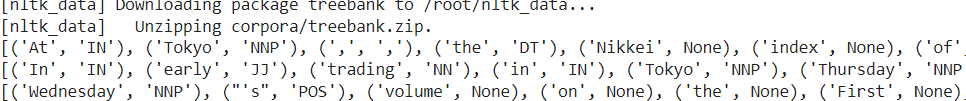
test\_sents = treebank.sents()[3000:3010]

for sent in test\_sents:

tagged\_sent = bigram\_tagger.tag(sent)

print(tagged\_sent)

**OUTPUT:**



1. Write program using the NLTK library to perform part-of-speech tagging on a text.

**PROGRAM:**

import nltk

def pos\_tagging(text):

words = nltk.word\_tokenize(text)

pos\_tags = nltk.pos\_tag(words)

return pos\_tags

text = "Good Morning students"

text1="Pay attention please"

text2="He is the founder of our university"

tags = pos\_tagging(text)

print(tags)

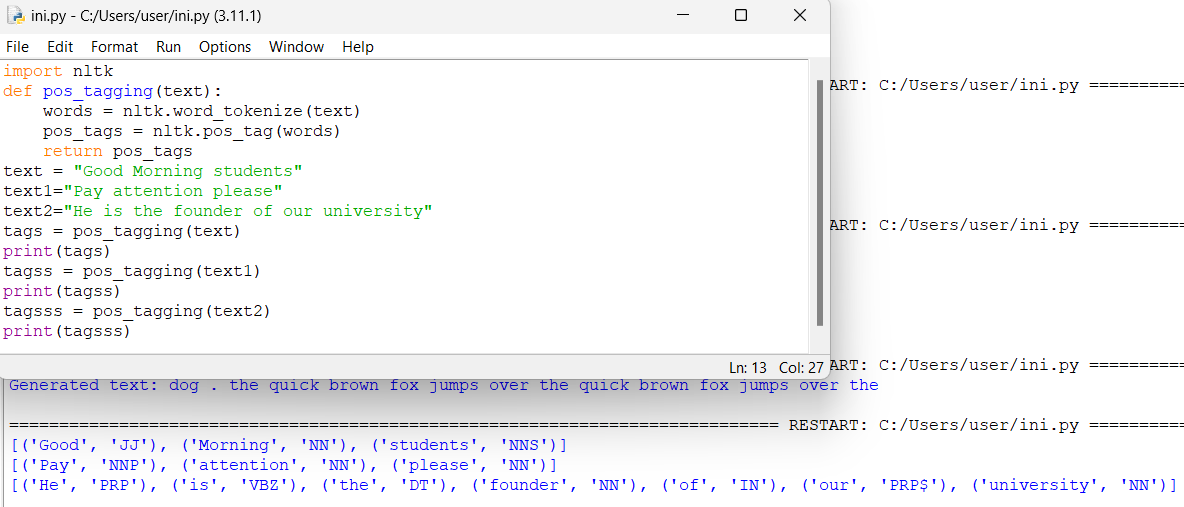
tagss = pos\_tagging(text1)

print(tagss)

tagsss = pos\_tagging(text2)

print(tagsss)

**OUTPUT:**



1. Implement a simple stochastic part-of-speech tagging algorithm using a basic probabilistic model to assign POS tags using python.

**PROGRAM:**

corpus = [

(["the", "cat", "sat"], ["DET", "NOUN", "VERB"]),

(["the", "dog", "barked"], ["DET", "NOUN", "VERB"]),

(["a", "cat", "meowed"], ["DET", "NOUN", "VERB"]),

(["the", "dog", "ran"], ["DET", "NOUN", "VERB"])

]

word\_tag\_probs = {}

for sentence, tags in corpus:

for word, tag in zip(sentence, tags):

if word not in word\_tag\_probs:

word\_tag\_probs[word] = {}

if tag not in word\_tag\_probs[word]:

word\_tag\_probs[word][tag] = 0

word\_tag\_probs[word][tag] += 1

for word in word\_tag\_probs:

total = sum(word\_tag\_probs[word].values())

for tag in word\_tag\_probs[word]:

word\_tag\_probs[word][tag] /= total

def simple\_pos\_tag(sentence):

tags = []

for word in sentence:

if word in word\_tag\_probs:

tag = max(word\_tag\_probs[word], key=word\_tag\_probs[word].get)

else:

tag = "NOUN"

tags.append(tag)

return tags

new\_sentence = ["the", "cat", "ran"]

tags = simple\_pos\_tag(new\_sentence)

print(list(zip(new\_sentence,tags)))

**OUTPUT:**



1. Implement a rule-based part-of-speech tagging system using regular expressions using python.

**PROGRAM:**

import re

rules = [

(r'\bthe\b', 'DET'),

(r'\ba\b', 'DET'),

(r'\ban\b', 'DET'),

(r'\b(cat|dog)\b', 'NOUN'),

(r'\b(sat|barked|meowed|ran)\b', 'VERB'),

(r'\b\w+ly\b', 'ADV'),

(r'\b\w+ing\b', 'VERB'),

(r'\b\w+ed\b', 'VERB'),

(r'\b\w+s\b', 'NOUN'),

]

def rule\_based\_pos\_tag(sentence):

tags = []

words = sentence.split()

for word in words:

tag = 'NOUN'

for pattern, rule\_tag in rules:

if re.fullmatch(pattern, word):

tag = rule\_tag

break

tags.append((word, tag))

return tags

sentence = "the cat sat on the mat and the dog barked"

tagged\_sentence = rule\_based\_pos\_tag(sentence)

print(tagged\_sentence)

**OUTPUT:**



1. Implement transformation-based tagging using a set of transformation rules, apply a simple rule to tag words using python.

**PROGRAM:**

corpus = [

["I", "want", "to", "book", "a", "flight"],

["the", "book", "is", "on", "the", "table"]

]

tagged\_corpus = [[(word, "NOUN") for word in sentence] for sentence in corpus]

rules = [

("NOUN", "VERB", lambda word, prev: word == "book" and prev == "to"),

]

def apply\_rules(tagged\_sentence, rules):

return [(word, next((new\_tag for current\_tag, new\_tag, condition in rules if tag == current\_tag and condition(word, tagged\_sentence[i-1][0] if i > 0 else None)), tag)) for i, (word, tag) in enumerate(tagged\_sentence)]

transformed\_corpus = [apply\_rules(sentence, rules) for sentence in tagged\_corpus]

for sentence in transformed\_corpus:

print(sentence)

**OUTPUT:**



1. Implement a simple top-down parser for context-free grammars using python.

**PROGRAM:**

import nltk

from nltk import CFG, RecursiveDescentParser

grammar = CFG.fromstring("""

S -> NP VP

NP -> Det N | Det N PP | 'I'

VP -> V NP | VP PP

PP -> P NP

Det -> 'a' | 'the'

N -> 'man' | 'park' | 'dog' | 'telescope'

V -> 'saw' | 'walked'

P -> 'in' | 'with'

""")

parser = RecursiveDescentParser(grammar)

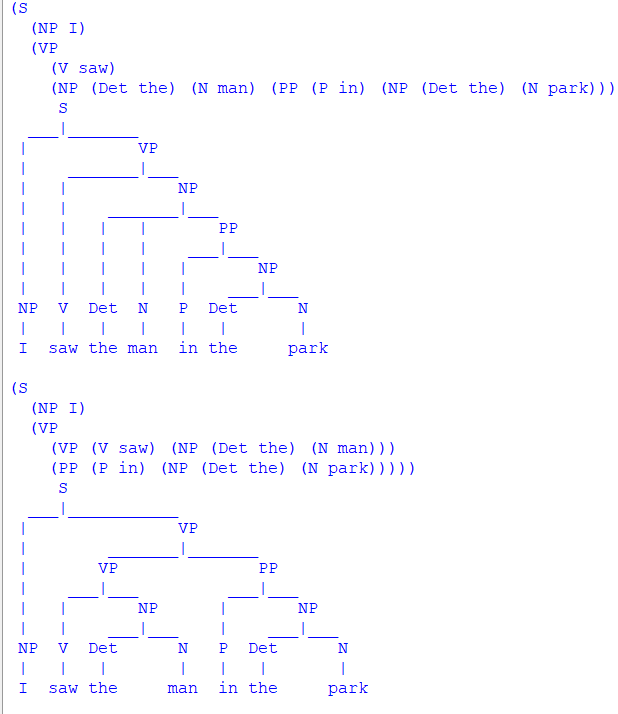
sentence = "I saw the man in the park".split()

for tree in parser.parse(sentence):

print(tree)

tree.pretty\_print()

**OUTPUT:**



1. Implement an Earley parser for context-free grammars using a simple python program.

**PROGRAM:**

class EarleyParser:

def \_\_init\_\_(self, grammar):

self.grammar = grammar

self.chart = []

def parse(self, tokens):

self.chart = [[] for \_ in range(len(tokens) + 1)]

start\_rule = self.grammar[0]

start\_state = (start\_rule[0], [], start\_rule[1], 0)

self.chart[0].append(start\_state)

for i in range(len(tokens) + 1):

for state in self.chart[i]:

if self.is\_complete(state):

self.completer(state, i)

elif self.is\_next\_non\_terminal(state):

self.predictor(state, i)

else:

if i < len(tokens):

self.scanner(state, i, tokens[i])

for state in self.chart[-1]:

if state[0] == start\_rule[0] and self.is\_complete(state) and state[3] == 0:

return True

return False

def is\_complete(self, state):

return len(state[1]) == len(state[2])

def is\_next\_non\_terminal(self, state):

return len(state[1]) < len(state[2]) and state[2][len(state[1])].isupper()

def predictor(self, state, index):

next\_non\_terminal = state[2][len(state[1])]

for rule in self.grammar:

if rule[0] == next\_non\_terminal:

new\_state = (rule[0], [], rule[1], index)

if new\_state not in self.chart[index]:

self.chart[index].append(new\_state)

def scanner(self, state, index, token):

next\_symbol = state[2][len(state[1])]

if next\_symbol == token:

new\_state = (state[0], state[1] + [token], state[2], state[3])

if new\_state not in self.chart[index + 1]:

self.chart[index + 1].append(new\_state)

def completer(self, state, index):

for old\_state in self.chart[state[3]]:

if len(old\_state[1]) < len(old\_state[2]) and old\_state[2][len(old\_state[1])] == state[0]:

new\_state = (old\_state[0], old\_state[1] + [state[0]], old\_state[2], old\_state[3])

if new\_state not in self.chart[index]:

self.chart[index].append(new\_state)

if \_\_name\_\_ == "\_\_main\_\_":

grammar = [

('S', ['NP', 'VP']),

('NP', ['Det', 'N']),

('VP', ['V', 'NP']),

('Det', ['the']),

('N', ['dog']),

('V', ['chases']),

]

parser = EarleyParser(grammar)

tokens = ['the', 'dog', 'chases', 'the', 'dog']

print(parser.parse(tokens))

**OUTPUT:**



1. Generate a parse tree for a given sentence using a context-free grammar using python program.

**PROGRAM:**

import nltk

from nltk import CFG, RecursiveDescentParser

grammar = CFG.fromstring("""

S -> NP VP

NP -> Det N | Det N PP | 'I'

VP -> V NP | VP PP

PP -> P NP

Det -> 'a' | 'the'

N -> 'man' | 'park' | 'dog' | 'telescope'

V -> 'saw' | 'walked'

P -> 'in' | 'with'

""")

parser = RecursiveDescentParser(grammar)

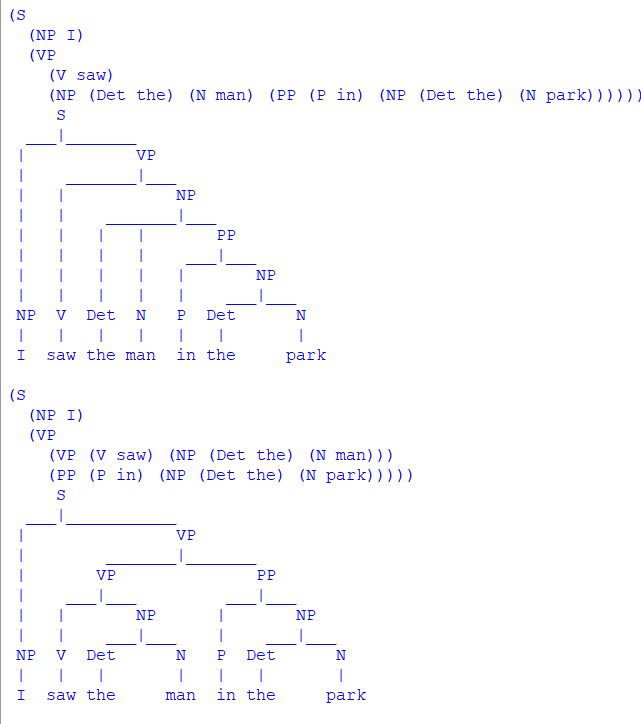
sentence = "I saw the man in the park".split()

for tree in parser.parse(sentence):

print(tree)

tree.pretty\_print()

**OUTPUT:**



1. Create a program in python to check for agreement in sentences based on a context-free

grammar's rules.

**PROGRAM:**

import nltk

from nltk import CFG

grammar = CFG.fromstring("""

S -> NP\_SG VP\_SG | NP\_PL VP\_PL

NP\_SG -> Det\_SG N\_SG

NP\_PL -> Det\_PL N\_PL

VP\_SG -> V\_SG

VP\_PL -> V\_PL

Det\_SG -> 'the'

Det\_PL -> 'the'

N\_SG -> 'cat' | 'dog'

N\_PL -> 'cats' | 'dogs'

V\_SG -> 'runs' | 'jumps'

V\_PL -> 'run' | 'jump'

""")

def check\_agreement(sentence):

tokens = sentence.split()

parser = nltk.ChartParser(grammar)

try:

next(parser.parse(tokens))

return True

except StopIteration:

return False

sentences = [

"the cat runs",

"the dogs run",

"the cat run",

"the dog runs"

]

for sentence in sentences:

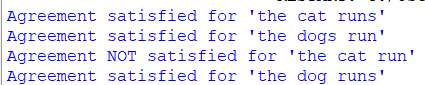
if check\_agreement(sentence):

print(f"Agreement satisfied for '{sentence}'")

else:

print(f"Agreement NOT satisfied for '{sentence}'")

**OUTPUT:**

****

1. Implement probabilistic context-free grammar parsing for a sentence using python.

**PROGRAM:**

import nltk

from nltk import PCFG, ViterbiParser

grammar = PCFG.fromstring("""

S -> NP VP [1.0]

VP -> V NP [0.7] | V [0.3]

NP -> Det N [0.6] | N [0.4]

Det -> 'the' [0.8] | 'a' [0.2]

N -> 'cat' [0.5] | 'dog' [0.5]

V -> 'chased' [0.9] | 'saw' [0.1]

""")

def parse\_sentence\_pcfg(sentence):

tokens = sentence.split()

parser = ViterbiParser(grammar)

try:

trees = list(parser.parse(tokens))

return trees[0]

except IndexError:

return None # Handle case where no parse tree is found

sentences = [

"the cat chased the dog",

"a dog saw the cat"

]

for sentence in sentences:

parse\_tree = parse\_sentence\_pcfg(sentence)

if parse\_tree:

print(f"Parse tree for '{sentence}':")

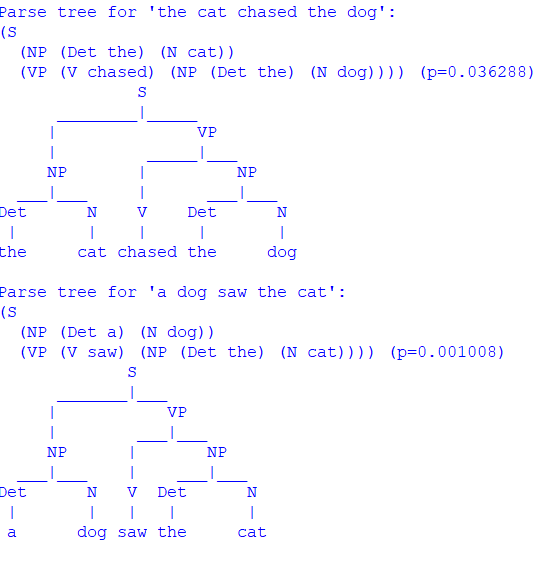
print(parse\_tree)

parse\_tree.pretty\_print()

else:

print(f"No parse tree found for sentence: '{sentence}'")

**OUTPUT:**



1. Implement a Python program using the SpaCy library to perform Named Entity Recognition (NER) on a given text.

**PROGRAM:**

import spacy

nlp = spacy.load('en\_core\_web\_sm')

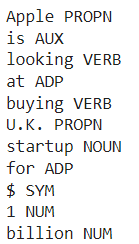
text = "Apple is looking at buying U.K. startup for $1 billion"

doc = nlp(text)

for token in doc:

    print(token.text, token.pos\_)

**OUTPUT:**



1. Write program demonstrates how to access WordNet, a lexical database, to retrieve synsets and explore word meanings in python.

**PROGRAM:**

import nltk

from nltk.corpus import wordnet

def explore\_word\_meanings(word):

synsets = wordnet.synsets(word)

if not synsets:

print(f"No meanings found for '{word}' in WordNet.")

return

for synset in synsets:

print(f"Meaning ({synset.pos()}) - {synset.definition()}")

print(f"Example: {synset.examples()}")

print()

word\_to\_explore = "bank"

explore\_word\_meanings(word\_to\_explore)

**OUTPUT:**

A close up of text

Description automatically generated

1. Implement a simple FOPC parser for basic logical expressions using python program.

**PROGRAM:**

18th code First Order Predicate Logic:

!pip install transformers

!pip install torch

import nltk

from nltk import CFG

fopc\_grammar = CFG.fromstring("""

S -> EXPR

EXPR -> PRED | EXPR 'AND' EXPR | EXPR 'OR' EXPR

PRED -> NAME '(' VARS ')'

VARS -> VAR | VAR ',' VARS

NAME -> 'P' | 'Q' | 'R'

VAR -> 'x' | 'y' | 'z'

AND -> 'AND'

OR -> 'OR'

""")

parser = nltk.ChartParser(fopc\_grammar)

def parse\_expression(expression):

try:

tokens = expression.replace('(', ' ( ').replace(')', ' ) ').replace(',', ' , ').split()

trees = list(parser.parse(tokens))

if trees:

for tree in trees:

tree.pretty\_print()

else:

print("No valid parse found.")

except ValueError as e:

print(f"Error: {e}")

expressions = [

"P(x,y)",

"Q(x) AND R(y,z)",

"P(x) OR Q(y) AND R(z)"

]

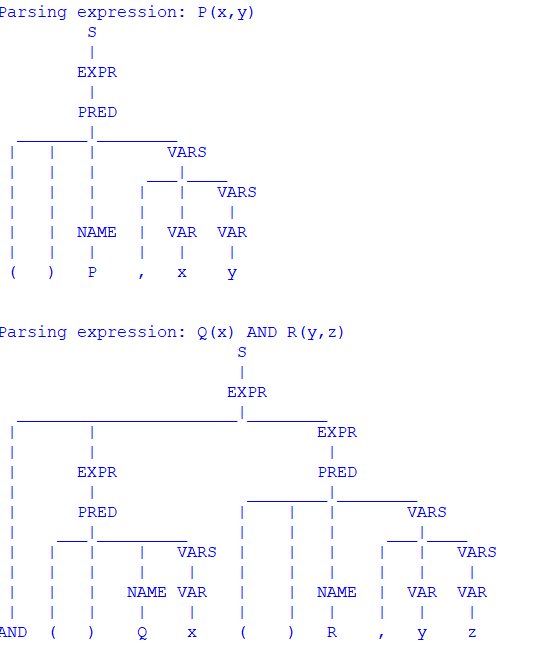
for expr in expressions:

print(f"Parsing expression: {expr}")

parse\_expression(expr)

print()

**OUTPUT:**

****

1. Create a program for word sense disambiguation using the Lesk algorithm using python.

**PROGRAM:**

from nltk.corpus import stopwords, wordnet

from nltk.tokenize import word\_tokenize

from nltk.wsd import lesk

import string

def preprocess\_text(text):

tokens = word\_tokenize(text.lower())

tokens = [token for token in tokens if token not in string.punctuation and token not in stopwords.words('english')]

return tokens

def perform\_wsd(word, sentence):

best\_sense = lesk(sentence, word)

return best\_sense

sentence = "He went to the bank to deposit his money."

word = "bank"

preprocessed\_sentence = preprocess\_text(sentence)

sense = perform\_wsd(word, preprocessed\_sentence)

if sense:

print(f"Word: {word}")

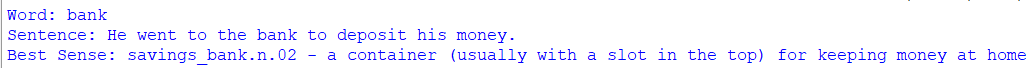
print(f"Sentence: {sentence}")

print(f"Best Sense: {sense.name()} - {sense.definition()}")

else:

print(f"No suitable sense found for '{word}' in the context.")

**OUTPUT:**



1. Implement a basic information retrieval system using TF-IDF (Term Frequency-Inverse Document Frequency) for document ranking using python.

**PROGRAM:**

import math

from nltk.corpus import stopwords

from nltk.tokenize import word\_tokenize

from nltk.probability import FreqDist

import string

documents = {

'doc1': "TF-IDF is a technique used in information retrieval and text mining.",

'doc2': "The TF-IDF score is used to determine the importance of a term within a document.",

'doc3': "Information retrieval is the process of obtaining information from a collection of text documents."

}

def preprocess\_text(text):

tokens = word\_tokenize(text.lower())

tokens = [token for token in tokens if token not in string.punctuation and token not in stopwords.words('english')]

return tokens

def compute\_tf(document):

fd = FreqDist(document)

tf = {word: fd[word] / len(document) for word in fd}

return tf

def compute\_idf(corpus):

all\_words = set(word for doc in corpus for word in doc)

idf = {word: math.log(len(corpus) / (1 + sum(1 for doc in corpus if word in doc))) for word in all\_words}

return idf

def compute\_tfidf(documents, idf):

tfidf\_scores = {}

for doc\_id, doc\_text in documents.items():

tokens = preprocess\_text(doc\_text)

tf = compute\_tf(tokens)

tfidf\_scores[doc\_id] = sum(tf[word] \* idf.get(word, 0) for word in tf)

return tfidf\_scores

def rank\_documents(query, documents):

preprocessed\_query = preprocess\_text(query)

idf = compute\_idf([preprocessed\_query] + [preprocess\_text(doc) for doc in documents.values()])

tfidf\_scores = compute\_tfidf(documents, idf)

ranked\_documents = sorted(tfidf\_scores.items(), key=lambda x: x[1], reverse=True)

return ranked\_documents

query = "TF-IDF information retrieval"

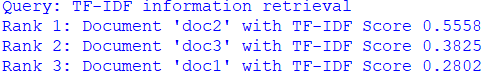
ranked\_docs = rank\_documents(query, documents)

print(f"Query: {query}")

for idx, (doc\_id, score) in enumerate(ranked\_docs, start=1):

print(f"Rank {idx}: Document '{doc\_id}' with TF-IDF Score {score:.4f}")

**OUTPUT:**

****

1. Create a python program that performs syntax-driven semantic analysis by extracting noun

phrases and their meanings from a sentence.

**PROGRAM:**

import nltk

from nltk.corpus import wordnet as wn

nltk.download('averaged\_perceptron\_tagger')

nltk.download('wordnet')

nltk.download('punkt')

def get\_noun\_phrases(sentence):

words = nltk.word\_tokenize(sentence)

pos\_tags = nltk.pos\_tag(words)

noun\_phrases = []

for word, pos in pos\_tags:

if pos.startswith('NN'): noun\_phrases.append(word)

return noun\_phrases

def get\_meaning(word):

synsets = wn.synsets(word, pos=wn.NOUN)

if synsets:

return synsets[0].definition()

return None

sentence = "The quick brown fox jumps over the lazy dog."

noun\_phrases = get\_noun\_phrases(sentence)

for noun in noun\_phrases:

meaning = get\_meaning(noun)

if meaning:

print(f"Noun: {noun} -> Meaning: {meaning}")

**OUTPUT:**



1. Create a python program that performs reference resolution within a text.

**PROGRAM**:

from nltk.tokenize import word\_tokenize, sent\_tokenize

from nltk.tag import pos\_tag

text = "John went to the store. He bought some milk."

sentences = sent\_tokenize(text)

tokenized\_sentences = [word\_tokenize(sentence) for sentence in sentences]

tagged\_sentences = [pos\_tag(sentence) for sentence in tokenized\_sentences]

def resolve\_references(tagged\_sentences):

resolved\_text = []

prev\_noun = None

for sentence in tagged\_sentences:

new\_sentence = []

for word, tag in sentence:

if tag in ['NN', 'NNP'] and prev\_noun is None:

prev\_noun = word

if word.lower() == 'he' and prev\_noun:

new\_sentence.append(prev\_noun)

else:

new\_sentence.append(word)

resolved\_text.append(" ".join(new\_sentence))

return " ".join(resolved\_text)

resolved\_text = resolve\_references(tagged\_sentences)

print("Original Text: ", text)

print("Resolved Text: ", resolved\_text)

**OUTPUT:**



1. Develop a python program that evaluates the coherence of a given text.

**PROGRAM:**

import nltk

from nltk.tokenize import word\_tokenize, sent\_tokenize

from nltk.corpus import stopwords

import string

nltk.download('punkt')

nltk.download('stopwords')

def preprocess\_text(text):

stop\_words = set(stopwords.words('english') + list(string.punctuation))

tokens = word\_tokenize(text.lower())

return [word for word in tokens if word not in stop\_words]

def lexical\_cohesion\_score(tokens):

score = 0

seen\_tokens = set()

for token in tokens:

if token in seen\_tokens:

score += 1

seen\_tokens.add(token)

return score / len(tokens) if tokens else 0

text = "John went to the store. He bought some milk. The store was busy."

tokens = preprocess\_text(text)

cohesion\_score = lexical\_cohesion\_score(tokens)

print("Text: ", text)

print("Lexical Cohesion Score: ", cohesion\_score)

**OUTPUT:**

****

1. Create a python program that recognizes dialog acts in a given dialog or conversation.

**PROGRAM:**

import nltk

from nltk.tokenize import word\_tokenize

dialog = ["Hi, how are you?", "I'm good, thank you!", "What's your name?", "My name is John."]

def recognize\_dialog\_act(utterance):

tokens = word\_tokenize(utterance.lower())

if tokens[0] in ['hi', 'hello']:

return 'Greeting'

elif tokens[-1] == '?':

return 'Question'

elif tokens[0] in ['thanks', 'thank']:

return 'Thanks'

else:

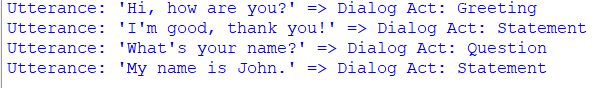
return 'Statement'

for utterance in dialog:

act = recognize\_dialog\_act(utterance)

print(f"Utterance: '{utterance}' => Dialog Act: {act}")

**OUTPUT:**

****

1. Utilize the GPT-3 model to generate text based on a given prompt. Make sure to install the OpenAI GPT-3 library in python implementation.

**PROGRAM:**

text generated prompt :

from transformers import pipeline

generator = pipeline('text-generation', model='gpt2', framework='pt')

def generate\_text(prompt, max\_length=100):

response = generator(prompt, max\_length=max\_length, num\_return\_sequences=1)

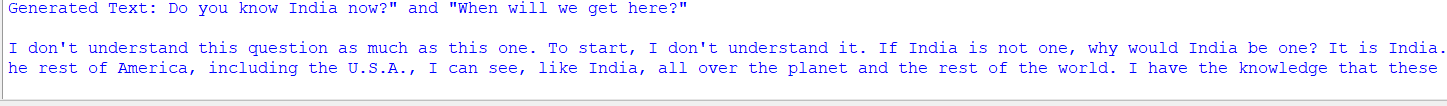
return response[0]['generated\_text']

prompt = "Do you know India"

generated\_text = generate\_text(prompt)

print(f"Generated Text: {generated\_text}")

**OUTPUT:**

****

1. Implement a machine translation program using the Hugging Face Transformers library, translate English text to French using python.

**PROGRAM:**

from transformers import MarianMTModel, MarianTokenizer

model\_name = 'Helsinki-NLP/opus-mt-en-fr'

tokenizer = MarianTokenizer.from\_pretrained(model\_name)

model = MarianMTModel.from\_pretrained(model\_name)

def translate(text, src\_lang="en", tgt\_lang="fr"):

translated = model.generate(\*\*tokenizer(text, return\_tensors="pt", padding=True))

translated\_text = [tokenizer.decode(t, skip\_special\_tokens=True) for t in translated]

return translated\_text[0]

if \_\_name\_\_ == "\_\_main\_\_":

english\_text = "Hello, how are you?"

french\_translation = translate(english\_text)

print(f"English: {english\_text}")

print(f"French: {french\_translation}")

**OUTPUT:**

****