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

**Program:**

import re

def main():

text = input("Enter some text: ")

pattern = r'\b[Aa]\w+'

matches = re.findall(pattern, text)

print("Words starting with 'A' or 'a':")

for match in matches:

print(match)

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

main()

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:**

def match(string):

state = 0

for char in string:

if state == 0 and char == 'a':

state = 1

elif state == 1 and char == 'b':

state = 2

return state == 2

print(match("hello world"))

print(match("ab"))

print(match("aab"))

print(match("abab"))

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

**Program:**

import nltk

from nltk.corpus import wordnet

def morphological\_analysis(word):

synsets = wordnet.synsets(word)

if synsets:

for synset in synsets:

print("Word:", synset.name())

print("POS:", synset.pos())

print("Definition:", synset.definition())

print("Examples:", synset.examples())

print()

else:

print("No morphological analysis found for the word:", word)

def main():

word = input("Enter a word to perform morphological analysis: ")

morphological\_analysis(word)

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

nltk.download('wordnet')

main()

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:**

import re

def pluralize(word):

rules = [

['[sxz]$', '$', 'es'],

['[^aeioudgkprt]h$', '$', 'es'],

['(qu|[^aeiou])y$', 'y$', 'ies'],

['$', '$', 's']

]

for rule in rules:

pattern, search, replace = rule

if re.search(pattern, word):

return re.sub(search, replace, word)

print(pluralize("cat"))

print(pluralize("dog"))

print(pluralize("knife"))

print(pluralize("potato"))

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

**Program:**

from nltk.stem import PorterStemmer

def stem\_words(words):

stemmer = PorterStemmer()

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

return stemmed\_words

def main():

words = input("Enter a list of words separated by spaces: ").split()

stemmed\_words = stem\_words(words)

print("Original words:", words)

print("Stemmed words:", stemmed\_words)

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

main()

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

**Program:**

import random

def generate\_bigram\_model(text):

words = text.split()

bigrams = [(words[i], words[i + 1]) for i in range(len(words) - 1)]

model = {}

for word1, word2 in bigrams:

if word1 in model:

model[word1].append(word2)

else:

model[word1] = [word2]

return model

def generate\_text\_bigram(model, num\_words=50):

current\_word = random.choice(list(model.keys()))

text = current\_word

for \_ in range(num\_words - 1):

if current\_word in model:

next\_word = random.choice(model[current\_word])

text += " " + next\_word

current\_word = next\_word

else:

break

return text

text = input("Enter your text: ")

model = generate\_bigram\_model(text)

generated\_text = generate\_text\_bigram(model)

print("Generated text:")

print(generated\_text)

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

**Program:**

import nltk

from nltk.tokenize import word\_tokenize

from nltk import pos\_tag

nltk.download('punkt')

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

def pos\_tagging(text):

words = word\_tokenize(text)

tagged\_words = pos\_tag(words)

return tagged\_words

text = input("Enter your text: ")

tagged\_text = pos\_tagging(text)

print("Tagged text:")

print(tagged\_text)

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

**Program:**

import random

class StochasticPOSTagger:

def \_\_init\_\_(self, training\_corpus):

self.word\_tag\_probabilities = self.calculate\_probabilities(training\_corpus)

def calculate\_probabilities(self, training\_corpus):

word\_tag\_counts = {}

tag\_counts = {}

for sentence in training\_corpus:

for word, tag in sentence:

word\_tag\_counts[(word, tag)] = word\_tag\_counts.get((word, tag), 0) + 1

tag\_counts[tag] = tag\_counts.get(tag, 0) + 1

word\_tag\_probabilities = {pair: count / tag\_counts[pair[1]] for pair, count in word\_tag\_counts.items()}

return word\_tag\_probabilities

def tag\_sentence(self, sentence):

tagged\_sentence = []

for word in sentence:

possible\_tags = [tag for (w, tag) in self.word\_tag\_probabilities.keys() if w == word]

if possible\_tags:

selected\_tag = random.choice(possible\_tags)

else:

selected\_tag = 'NOUN'

tagged\_sentence.append((word, selected\_tag))

return tagged\_sentence

training\_corpus = [

[('The', 'DET'), ('cat', 'NOUN'), ('is', 'VERB'), ('on', 'PREP'), ('the', 'DET'), ('mat', 'NOUN')],

[('A', 'DET'), ('dog', 'NOUN'), ('is', 'VERB'), ('running', 'VERB')]

]

tagger = StochasticPOSTagger(training\_corpus)

new\_sentence = ['The', 'dog', 'is', 'running']

tagged\_sentence = tagger.tag\_sentence(new\_sentence)

print(tagged\_sentence)

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

**Program:**

import nltk

from nltk import pos\_tag, word\_tokenize

def perform\_pos\_tagging(text):

words = word\_tokenize(text)

tagged\_words = pos\_tag(words)

return tagged\_words

text = "NLTK is a powerful library for natural language processing."

tagged\_words = perform\_pos\_tagging(text)

print("Original Text:", text)

print("Part-of-Speech Tagging Result:", tagged\_words)

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

**Program:**

import nltk

def apply\_rule(word, tag, rules):

for rule in rules:

if word in rule[0]:

if tag in rule[1]:

return rule[1]

return tag

rules = [

(r'\d+', 'CD'),

(r'[a-zA-Z]+(ed|ing|es|)?$', 'VB'),

(r'[a-zA-Z]+(ly|ment)?$', 'RB'),

(r'[a-zA-Z]+(able|ible)?$', 'JJ'),

(r'\b(a|an|the)\b', 'DT'),

(r'[a-zA-Z]+', 'NN')

]

text = "The quick brown fox jumps over the lazy dog"

words = nltk.word\_tokenize(text)

tagged\_words = []

for word in words:

tag = apply\_rule(word, 'NN', rules)

tagged\_words.append((word, tag))

print(tagged\_words)

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

**Program:**

class Parser:

def \_\_init\_\_(self, input\_string):

self.tokens = input\_string.split()

self.current\_token = None

self.index = 0

def get\_next\_token(self):

if self.index < len(self.tokens):

self.current\_token = self.tokens[self.index]

self.index += 1

else:

self.current\_token = None

def parse(self):

self.get\_next\_token()

self.expression()

def expression(self):

self.term()

while self.current\_token in ['+', '-']:

operator = self.current\_token

self.get\_next\_token()

self.term()

print(f' {operator} ', end='')

def term(self):

self.factor()

while self.current\_token in ['\*', '/']:

operator = self.current\_token

self.get\_next\_token()

self.factor()

print(f' {operator} ', end='')

def factor(self):

if self.current\_token.isdigit():

print(self.current\_token, end='')

self.get\_next\_token()

elif self.current\_token == '(':

self.get\_next\_token()

self.expression()

if self.current\_token == ')':

self.get\_next\_token()

else:

raise SyntaxError("Expected ')'")

else:

raise SyntaxError("Invalid token")

input\_string = "3 + 4 \* ( 2 - 1 )"

parser = Parser(input\_string)

parser.parse()

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

**Program:**

class State:

def \_\_init\_\_(self, rule, dot\_index, start\_index):

self.rule = rule

self.dot\_index = dot\_index

self.start\_index = start\_index

def \_\_eq\_\_(self, other):

return self.rule == other.rule and self.dot\_index == other.dot\_index and self.start\_index == other.start\_index

def \_\_hash\_\_(self):

return hash((self.rule, self.dot\_index, self.start\_index))

def \_\_str\_\_(self):

return f'{self.rule} : {"".join(self.rule)} - {self.dot\_index} - {self.start\_index}'

def earley\_parse(grammar, sentence):

chart = [[] for \_ in range(len(sentence) + 1)]

start\_rule = next(iter(grammar))

start\_state = State(start\_rule, 0, 0)

chart[0].append(start\_state)

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

while True:

added = False

for state in chart[i]:

if state.dot\_index < len(state.rule) and state.rule[state.dot\_index] in grammar:

non\_terminal = state.rule[state.dot\_index]

for rule in grammar[non\_terminal]:

new\_state = State(rule, 0, i)

if new\_state not in chart[i]:

chart[i].append(new\_state)

added = True

elif state.dot\_index < len(state.rule) and state.rule[state.dot\_index] == sentence[i - 1]:

new\_state = State(state.rule, state.dot\_index + 1, state.start\_index)

if new\_state not in chart[i]:

chart[i].append(new\_state)

added = True

elif state.dot\_index == len(state.rule):

for s in chart[state.start\_index]:

if s.dot\_index < len(s.rule) and s.rule[s.dot\_index] == state.rule[0]:

new\_state = State(s.rule, s.dot\_index + 1, s.start\_index)

if new\_state not in chart[i]:

chart[i].append(new\_state)

added = True

if not added:

break

for i in range(len(chart)):

print(f"Chart[{i}]:")

for state in chart[i]:

print(state)

grammar = {

'S': [['NP', 'VP']],

'NP': [['DET', 'N'], ['N']],

'VP': [['V', 'NP']],

'DET': ['the', 'a'],

'N': ['man', 'ball', 'woman', 'table'],

'V': ['hit', 'took', 'saw', 'liked']

}

sentence = ['the', 'man', 'hit', 'the', 'table']

earley\_parse(grammar, sentence)

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

**Program:**

import nltk

def generate\_parse\_tree(sentence, grammar):

parser = nltk.ChartParser(grammar)

tokens = nltk.word\_tokenize(sentence)

parse\_trees = list(parser.parse(tokens))

return parse\_trees

simple\_grammar = nltk.CFG.fromstring("""

S -> NP VP

NP -> Det N

VP -> V NP

Det -> 'the' | 'a'

N -> 'dog' | 'cat'

V -> 'chased' | 'caught'

""")

sentence = "the dog chased a cat"

parse\_trees = generate\_parse\_tree(sentence, simple\_grammar)

for i, tree in enumerate(parse\_trees):

print(f"Parse Tree {i + 1}:")

tree.pretty\_print()

print()

**14.      Create a program in python to check for agreement in sentences based on a context-free grammar's rules.**

**Program:**

import nltk

def check\_agreement(sentence, grammar):

parser = nltk.ChartParser(grammar)

tokens = nltk.word\_tokenize(sentence)

try:

parse\_tree = next(parser.parse(tokens))

return True

except StopIteration:

return False

agreement\_grammar = nltk.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'

N\_PL -> 'cats'

V\_SG -> 'chases'

V\_PL -> 'chase'

""")

sentence1 = "the cat chases"

sentence2 = "the cats chases"

result1 = check\_agreement(sentence1, agreement\_grammar)

result2 = check\_agreement(sentence2, agreement\_grammar)

print(f"Sentence 1 Agreement: {'Yes' if result1 else 'No'}")

print(f"Sentence 2 Agreement: {'Yes' if result2 else 'No'}")

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

**Program:**

import nltk

pcfg\_grammar = nltk.PCFG.fromstring("""

S -> NP VP [1.0]

NP -> Det N [0.5] | N [0.3] | N PP [0.2]

VP -> V NP [0.9] | VP PP [0.1]

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

N -> 'dog' [0.4] | 'cat' [0.3] | 'park' [0.3]

V -> 'chased' [0.7] | 'caught' [0.3]

PP -> P NP [1.0]

P -> 'in' [0.6] | 'on' [0.4]

""")

pcfg\_parser = nltk.ViterbiParser(pcfg\_grammar)

sentence = "the dog chased the cat in the park"

tokens = nltk.word\_tokenize(sentence)

for tree in pcfg\_parser.parse(tokens):

tree.pretty\_print()

print("Probability:", tree.prob())

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

**Program:**

import spacy

def perform\_ner(text):

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

doc = nlp(text)

for ent in doc.ents:

print(f"Entity: {ent.text}, Label: {ent.label\_}")

text = "Apple Inc. is an American multinational technology company headquartered in Cupertino, California."

perform\_ner(text)

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

**Program:**

import nltk

# Download WordNet data

nltk.download('wordnet')

from nltk.corpus import wordnet

def explore\_word\_meanings(word):

synsets = wordnet.synsets(word)

if not synsets:

print(f"No synsets found for '{word}'.")

else:

print(f"Synsets for '{word}':")

for synset in synsets:

print(f" - {synset.name()} ({synset.pos()})")

print(f" Definition: {synset.definition()}")

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

print()

word\_to\_explore = "bottle"

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

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

**Program:**

from pyparsing import Word, alphanums, oneOf, infixNotation, opAssoc, Forward, ParseException

variable = Word(alphanums)

predicate = Word(alphanums)

quantifier = oneOf("forall exists")

expression = Forward()

unary\_operators = oneOf("not ¬")

quantifier\_operators = oneOf("forall exists")

atom = variable | predicate

expression << (

(quantifier + variable + "(" + expression + ")") |

(unary\_operators + expression) |

atom

)

def parse\_fopc(expression\_str):

try:

result = expression.parseString(expression\_str, parseAll=True)

return result[0]

except ParseException as e:

print(f"Error parsing expression: {e}")

return None

expression1 = "forall x (P(x) -> Q(x))"

expression2 = "exists y (P(y) & Q(y))"

parsed\_expression1 = parse\_fopc(expression1)

parsed\_expression2 = parse\_fopc(expression2)

print("Parsed Expression 1:", parsed\_expression1)

print("Parsed Expression 2:", parsed\_expression2)

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

**Program:**

from nltk.corpus import wordnet

from nltk.tokenize import word\_tokenize

from nltk.corpus import stopwords

import nltk

nltk.download('punkt')

nltk.download('stopwords')

nltk.download('wordnet')

def lesk(word, sentence):

sentence\_tokens = set(word\_tokenize(sentence.lower()))

best\_sense = None

max\_overlap = 0

for synset in wordnet.synsets(word):

signature = set(word\_tokenize(synset.definition().lower()))

signature.update(word\_tokenize(" ".join(synset.examples()).lower()))

overlap = len(sentence\_tokens.intersection(signature))

if overlap > max\_overlap:

max\_overlap = overlap

best\_sense = synset

return best\_sense

word = "bank"

sentence = "He sat on the bank of the river and watched the sunset."

sense = lesk(word, sentence)

if sense:

print("Word:", word)

print("Sentence:", sentence)

print("Best Sense:", sense)

print("Definition:", sense.definition())

else:

print("No sense found for the word:", word)

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

**Program:**

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.metrics.pairwise import linear\_kernel

documents = [

"TF-IDF, or term frequency-inverse document frequency, is a statistic that evaluates the importance of a word in a document relative to a collection of documents.",

"It is often used in information retrieval and text mining.",

"The TF-IDF score increases with the frequency of the word in the document and is offset by the frequency of the word in the entire collection of documents.",

"In other words, TF-IDF highlights words that are unique to a document and are not common across many documents.",

"To implement TF-IDF, you need to calculate the term frequency and inverse document frequency for each word in the document collection.",

]

query = "TF-IDF in information retrieval"

vectorizer = TfidfVectorizer()

tfidf\_matrix = vectorizer.fit\_transform(documents + [query])

cosine\_similarities = linear\_kernel(tfidf\_matrix[-1], tfidf\_matrix[:-1]).flatten()

document\_ranking = sorted(enumerate(cosine\_similarities), key=lambda x: x[1], reverse=True)

print("Ranked Documents:")

for idx, similarity in document\_ranking:

print(f"Document {idx + 1}: Similarity = {similarity:.4f}")

print(f" {documents[idx]}")

print()

**21. 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 import pos\_tag

from nltk.tokenize import word\_tokenize

from nltk.chunk import ne\_chunk

# Download NLTK resources if not already downloaded

nltk.download('punkt')

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

nltk.download('maxent\_ne\_chunker')

nltk.download('words')

def extract\_noun\_phrases(sentence):

words = word\_tokenize(sentence)

tagged\_words = pos\_tag(words)

chunked\_sentence = ne\_chunk(tagged\_words)

noun\_phrases = []

for subtree in chunked\_sentence:

if isinstance(subtree, nltk.tree.Tree) and subtree.label() == 'NP':

noun\_phrase = " ".join(word for word, tag in subtree.leaves())

noun\_phrases.append(noun\_phrase)

return noun\_phrases

def determine\_meaning(noun\_phrase):

# Dummy implementation, replace this with your actual logic to determine meanings

# For this example, we'll just return a placeholder meaning

meanings = {

"cat": "a small domesticated carnivorous mammal with soft fur, a short snout, and retractile claws",

"ball": "a round or spherical object that is used in games and sports"

}

return meanings.get(noun\_phrase, "Meaning not found")

def main():

sentence = input("Enter a sentence: ")

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

print("\nNoun phrases found in the sentence:")

for noun\_phrase in noun\_phrases:

print("- {}".format(noun\_phrase))

meaning = determine\_meaning(noun\_phrase.lower())

print(" Meaning:", meaning)

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

main()

**22. Create a python program that performs reference resolution within a text.**

**Program:**

import nltk

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

from nltk.tag import pos\_tag

from nltk.chunk import ne\_chunk

# Download NLTK resources if not already downloaded

nltk.download('punkt')

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

nltk.download('maxent\_ne\_chunker')

nltk.download('words')

def resolve\_references(text):

sentences = sent\_tokenize(text)

resolved\_text = ""

for sentence in sentences:

tagged\_sentence = pos\_tag(word\_tokenize(sentence))

chunked\_sentence = ne\_chunk(tagged\_sentence)

resolved\_sentence = ""

for subtree in chunked\_sentence:

if type(subtree) == nltk.tree.Tree:

if subtree.label() == 'NP': # Noun phrase

antecedent = find\_antecedent(subtree)

resolved\_sentence += antecedent + " "

else:

resolved\_sentence += " ".join(word for word, tag in subtree.leaves()) + " "

else:

resolved\_sentence += subtree[0] + " "

resolved\_text += resolved\_sentence.strip() + "\n"

return resolved\_text

def find\_antecedent(subtree):

for node in subtree:

if type(node) == nltk.tree.Tree:

if node.label() == 'NP':

return find\_antecedent(node)

else:

if node[1] == 'PRP':

antecedent = get\_antecedent(node)

return antecedent

return ""

def get\_antecedent(pronoun):

antecedent = ""

if pronoun[0].lower() == 'he':

antecedent = "John" # Replace with actual antecedent from context

elif pronoun[0].lower() == 'she':

antecedent = "Mary" # Replace with actual antecedent from context

# Add more cases for other pronouns as needed

return antecedent

def main():

text = """

John saw Mary at the store. She was buying groceries.

He decided to greet her.

"""

resolved\_text = resolve\_references(text)

print("Original text:")

print(text)

print("\nResolved text:")

print(resolved\_text)

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

main()

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

**Program:**

import nltk

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

from nltk.corpus import stopwords

from nltk.collocations import BigramAssocMeasures, BigramCollocationFinder

# Download NLTK resources if not already downloaded

nltk.download('punkt')

nltk.download('stopwords')

def calculate\_coherence(text):

sentences = sent\_tokenize(text)

word\_tokens = [word.lower() for sentence in sentences for word in word\_tokenize(sentence) if word.isalnum()]

filtered\_tokens = [word for word in word\_tokens if word not in stopwords.words('english')]

bigram\_measures = BigramAssocMeasures()

finder = BigramCollocationFinder.from\_words(filtered\_tokens)

pmi = finder.score\_ngrams(bigram\_measures.pmi)

coherence\_score = sum(score for bigram, score in pmi) / len(pmi)

return coherence\_score

def main():

text = """

Coherence in writing means that all the ideas in a paragraph flow smoothly from one sentence to the next.

With each sentence building on the previous one and leading logically to the next.

Achieving coherence requires paying attention to the organization and development of ideas.

"""

coherence\_score = calculate\_coherence(text)

print("Coherence score:", coherence\_score)

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

main()

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

**Program:**

import nltk

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

# Download NLTK resources if not already downloaded

nltk.download('punkt')

def recognize\_dialog\_acts(dialog):

sentences = sent\_tokenize(dialog)

dialog\_acts = []

for sentence in sentences:

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

dialog\_act = classify\_dialog\_act(tokens)

dialog\_acts.append((sentence, dialog\_act))

return dialog\_acts

def classify\_dialog\_act(tokens):

greetings = ['hello', 'hi', 'hey']

farewells = ['goodbye', 'bye', 'see you']

questions = ['what', 'where', 'when', 'who', 'why', 'how']

statements = ['i think', 'i believe', 'in my opinion']

if any(token in greetings for token in tokens):

return 'GREETING'

elif any(token in farewells for token in tokens):

return 'FAREWELL'

elif any(token in questions for token in tokens):

return 'QUESTION'

elif any(token in statements for token in tokens):

return 'STATEMENT'

else:

return 'OTHER'

def main():

dialog = """

Speaker 1: Hi, how are you?

Speaker 2: I'm doing well, thank you. How about you?

Speaker 1: Goodbye then, have a great day!

"""

dialog\_acts = recognize\_dialog\_acts(dialog)

print("Dialog Acts:")

for utterance, dialog\_act in dialog\_acts:

print(f"{utterance.strip()} - {dialog\_act}")

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

main()

**25. 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:**

import openai

# Set your OpenAI API key

api\_key = 'YOUR\_API\_KEY'

openai.api\_key = api\_key

def generate\_text(prompt, max\_tokens=50):

response = openai.Completion.create(

engine="text-davinci-002", # Choose the GPT-3 engine you prefer

prompt=prompt,

max\_tokens=max\_tokens

)

return response.choices[0].text.strip()

def main():

prompt = "Once upon a time"

generated\_text = generate\_text(prompt)

print("Generated text:")

print(generated\_text)

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

main()

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

**Program:**

from transformers import MarianMTModel, MarianTokenizer

def translate\_text(text, model\_name="Helsinki-NLP/opus-mt-en-fr"):

# Load pre-trained model and tokenizer

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

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

# Tokenize input text

inputs = tokenizer(text, return\_tensors="pt", truncation=True, padding=True)

# Perform translation

outputs = model.generate(\*\*inputs)

# Decode the translated text

translated\_text = tokenizer.decode(outputs[0], skip\_special\_tokens=True)

return translated\_text

def main():

# Input English text

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

# Translate English text to French

french\_text = translate\_text(english\_text)

# Output translated text

print("Translated French text:")

print(french\_text)

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

main()