**PROGRAM [1]:**

import random

class TicTacToe:

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

self.board = [['-' for \_ in range(3)] for \_ in range(3)]

def get\_random\_first\_player(self):

return random.choice(['X', 'O'])

def fix\_spot(self, row, col, player):

self.board[row][col] = player

def is\_player\_win(self, player):

n = len(self.board)

for i in range(n):

if all(self.board[i][j] == player for j in range(n)) or all(self.board[j][i] == player for j in range(n)):

return True

if all(self.board[i][i] == player for i in range(n)) or all(self.board[i][n - 1 - i] == player for i in range(n)):

return True

return False

def is\_board\_filled(self):

return all(item != '-' for row in self.board for item in row)

def swap\_player\_turn(self, player):

return 'X' if player == 'O' else 'O'

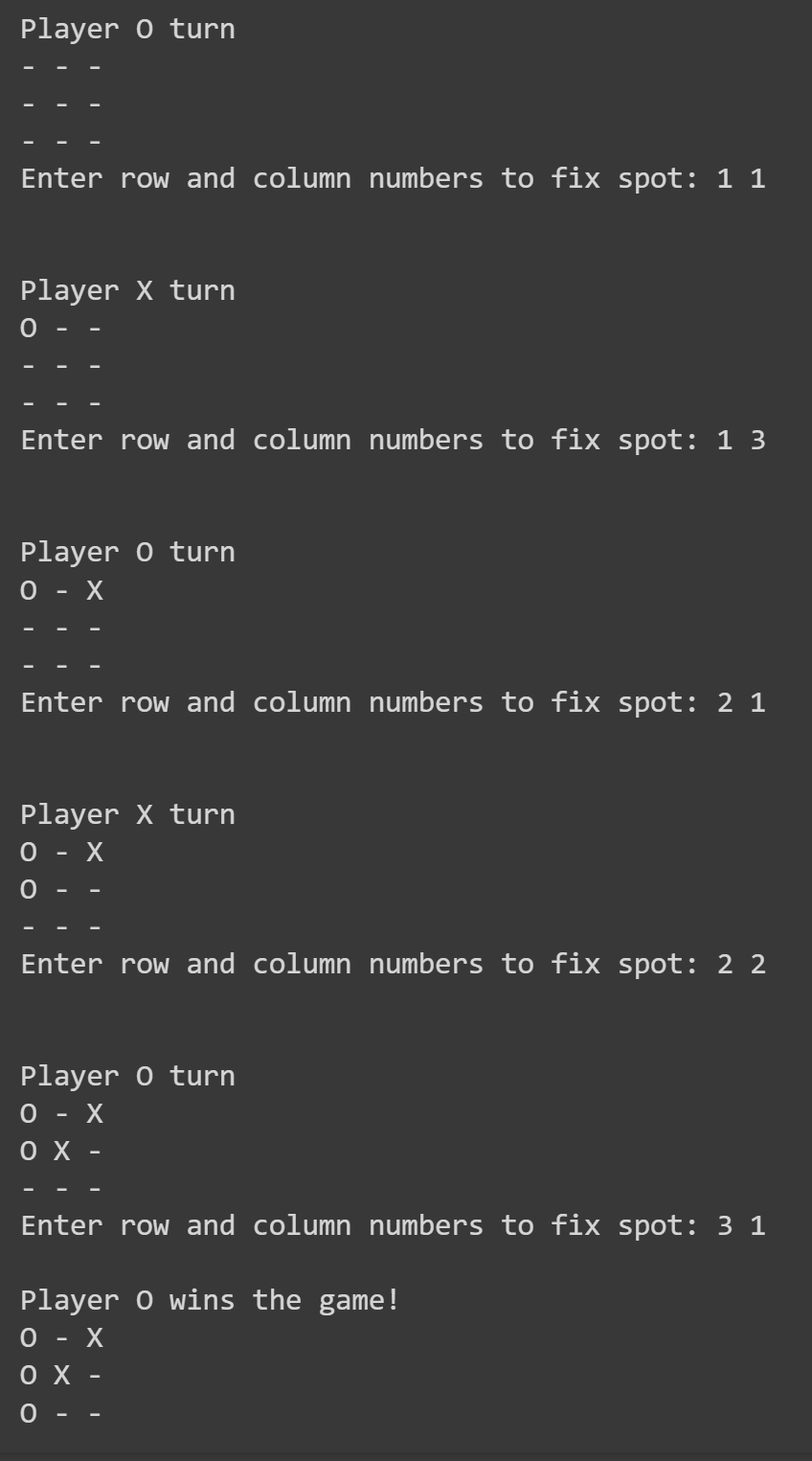
def show\_board(self):

for row in self.board:

print(\*row)

def start(self):

**OUTPUT [1]:**



self.get\_random\_first\_player()

player = 'X' if self.get\_random\_first\_player() == 'X' else 'O'

while True:

print(f"Player {player} turn")

self.show\_board()

row, col = map(int, input("Enter row and column numbers to fix spot: ").split())

print()

self.fix\_spot(row - 1, col - 1, player)

if self.is\_player\_win(player):

print(f"Player {player} wins the game!")

break

if self.is\_board\_filled():

print("Match Draw!")

break

player = self.swap\_player\_turn(player)

print()

self.show\_board()

tic\_tac\_toe = TicTacToe()

tic\_tac\_toe.start()

**PROGRAM [2]:**

import copy

from heapq import heappush, heappop

n = 3

rows = [1, 0, -1, 0]

cols = [0, -1, 0, 1]

class PriorityQueue:

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

self.heap = []

def push(self, key):

heappush(self.heap, key)

def pop(self):

return heappop(self.heap)

def empty(self):

return not bool(self.heap)

class Nodes:

def \_\_init\_\_(self, parent, mats, empty\_tile\_posi, costs, levels):

self.parent = parent

self.mats = mats

self.empty\_tile\_posi = empty\_tile\_posi

self.costs = costs

self.levels = levels

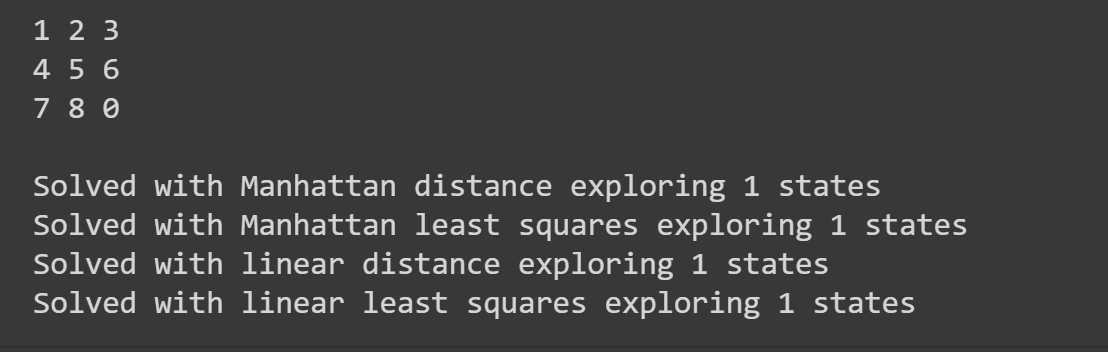
def \_\_lt\_\_(self, nxt):

return self.costs < nxt.costs

def calculate\_costs(mats, final):

return sum(1 for i in range(n) for j in range(n) if mats[i][j] and mats[i][j] != final[i][j])

**OUTPUT [2]:**



def new\_nodes(mats, empty\_tile\_posi, new\_empty\_tile\_posi, levels, parent, final):

new\_mats = copy.deepcopy(mats)

x1, y1 = empty\_tile\_posi

x2, y2 = new\_empty\_tile\_posi

new\_mats[x1][y1], new\_mats[x2][y2] = new\_mats[x2][y2], new\_mats[x1][y1]

costs = calculate\_costs(new\_mats, final)

return Nodes(parent, new\_mats, new\_empty\_tile\_posi, costs, levels)

def print\_matrix(mats):

for row in mats:

print(\*row)

print()

def is\_safe(x, y):

return 0 <= x < n and 0 <= y < n

def print\_path(root):

if root is None:

return

print\_path(root.parent)

print\_matrix(root.mats)

def solve(initial, empty\_tile\_posi, final):

pq = PriorityQueue()

costs = calculate\_costs(initial, final)

root = Nodes(None, initial, empty\_tile\_posi, costs, 0)

pq.push(root)

while not pq.empty():

minimum = pq.pop()

if minimum.costs == 0:

print\_path(minimum)

return

for i in range(n):

new\_tile\_posi = [minimum.empty\_tile\_posi[0] + rows[i], minimum.empty\_tile\_posi[1] + cols[i]]

if is\_safe(\*new\_tile\_posi):

child = new\_nodes(minimum.mats, minimum.empty\_tile\_posi, new\_tile\_posi, minimum.levels + 1, minimum, final)

pq.push(child)

initial = [[1, 2, 3], [5, 6, 0], [7, 8, 4]]

final = [[1, 2, 3], [5, 8, 6], [0, 7, 4]]

empty\_tile\_posi = [1, 2]

solve(initial, empty\_tile\_posi, final)

**PROGRAM [3]:**

global N

N = 4

def printSolution(board):

for i in range(N):

for j in range(N):

print (board[i][j],end=' ')

print()

def isSafe(board, row, col):

for i in range(col):

if board[row][i] == 1:

return False

for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

if board[i][j] == 1:

return False

for i, j in zip(range(row, N, 1), range(col, -1, -1)):

if board[i][j] == 1:

return False

return True

def solveNQUtil(board, col):

if col >= N:

return True

for i in range(N):

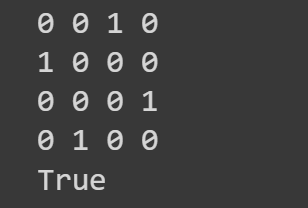
if isSafe(board, i, col):

board[i][col] = 1

if solveNQUtil(board, col + 1) == True:

return True

**OUTPUT [3]:**



board[i][col] = 0

return False

def solveNQ():

board = [ [0, 0, 0, 0],

[0, 0, 0, 0],

[0, 0, 0, 0],

[0, 0, 0, 0]

]

if solveNQUtil(board, 0) == False:

print ("Solution does not exist")

return False

printSolution(board)

return True

solveNQ()

**PROGRAM [4]:**

from collections import defaultdict

class Graph:

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

self.graph = defaultdict(list)

def add\_edge(self, u, v):

self.graph[u].append(v)

def dfs\_util(self, v, visited):

visited[v] = True

print(v, end=" ")

for i in self.graph[v]:

if not visited[i]:

self.dfs\_util(i, visited)

def dfs(self):

V = len(self.graph)

visited = [False] \* V

for i in range(V):

if not visited[i]:

self.dfs\_util(i, visited)

g = Graph()

g.add\_edge(0, 1)

g.add\_edge(0, 2)

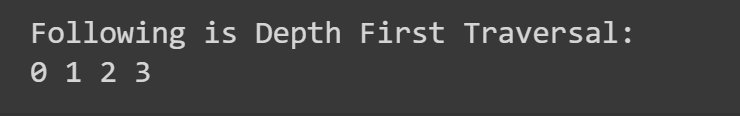
g.add\_edge(1, 2)

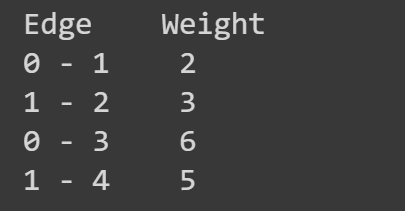
g.add\_edge(2, 0)

g.add\_edge(2, 3)

g.add\_edge(3, 3)

**OUTPUT [4]:**





print("Following is Depth First Traversal:")

g.dfs()

**PROGRAM [5]:**

graph = {

'A': {'B': 10, 'C': 20},

'B': {'A': 10, 'D': 5, 'E': 15},

'C': {'A': 20, 'F': 30},

'D': {'B': 5},

'E': {'B': 15, 'F': 5},

'F': {'C': 30, 'E': 5}

}

# Define the heuristic function for A\* algorithm

def heuristic(a, b):

return abs(ord(a) - ord(b))

# Define the BFS function

def bfs(graph, start, end):

queue = [(start, [start], 0)]

while queue:

node, path, cost = queue.pop(0)

for next\_node in graph[node]:

if next\_node == end:

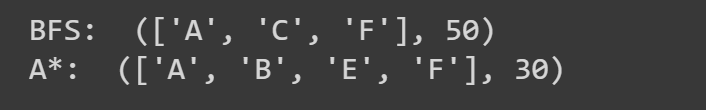
return path + [next\_node], cost + graph[node][next\_node]

else:

queue.append((next\_node, path + [next\_node], cost + graph[node][next\_node]))

# Define the A\* function

**OUTPUT [5]:**



def a\_star(graph, start, end):

queue = [(0, start, [start], 0)]

visited = set()

while queue:

f\_cost, node, path, cost = queue.pop(0)

if node in visited:

continue

visited.add(node)

if node == end:

return path, cost

for next\_node in graph[node]:

g\_cost = cost + graph[node][next\_node]

h\_cost = heuristic(next\_node, end)

queue.append((g\_cost + h\_cost, next\_node, path + [next\_node], g\_cost))

queue.sort(key=lambda x: x[0])

# Test the algorithms

print("BFS: ", bfs(graph, 'A', 'F'))

print("A\*: ", a\_star(graph, 'A', 'F'))

**PROGRAM [6]:**

import math

def minimax(curDepth, nodeIndex, maxTurn, scores, targetDepth):

if curDepth == targetDepth:

return scores[nodeIndex]

if maxTurn:

return max(minimax(curDepth + 1, nodeIndex \* 2, False, scores, targetDepth),

minimax(curDepth + 1, nodeIndex \* 2 + 1, False, scores, targetDepth))

else:

return min(minimax(curDepth + 1, nodeIndex \* 2, True, scores, targetDepth),

minimax(curDepth + 1, nodeIndex \* 2 + 1, True, scores, targetDepth))

scores = [3, 5, 2, 9, 12, 5, 23, 23]

treeDepth = int(math.log2(len(scores)))

print("The optimal value is:", end=" ")

print(minimax(0, 0, True, scores, treeDepth))

**OUTPUT [6]:**



**PROGRAM [7]:**

def get\_index\_comma(string):

index\_list = list()

par\_count = 0

for i in range(len(string)):

if string[i] == ',' and par\_count == 0:

index\_list.append(i)

elif string[i] == '(':

par\_count += 1

elif string[i] == ')':

par\_count -= 1

return index\_list

def is\_variable(expr):

for i in expr:

if i == '(':

return False

return True

def process\_expression(expr):

expr = expr.replace(' ', '') # Removed empty spaces

index = None

for i in range(len(expr)):

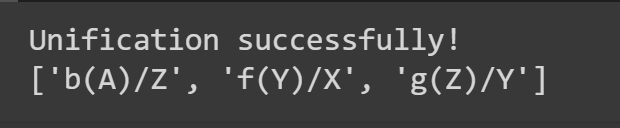
if expr[i] == '(':

index = i

break

predicate\_symbol = expr[:index]

**OUTPUT [7]:**



expr = expr.replace(predicate\_symbol, '')

expr = expr[1:len(expr)-1]

arg\_list = list()

indices = get\_index\_comma(expr)

if len(indices) == 0:

arg\_list.append(expr)

else:

arg\_list.append(expr[:indices[0]])

for i, j in zip(indices, indices[1:]):

arg\_list.append(expr[i+1:j])

arg\_list.append(expr[indices[len(indices)-1]+1:])

return predicate\_symbol, arg\_list

def get\_arg\_list(expr):

\_, arg\_list = process\_expression(expr)

flag = True

while flag:

flag = False

for i in arg\_list:

if not is\_variable(i):

flag = True

\_, tmp = process\_expression(i)

for j in tmp:

if j not in arg\_list:

arg\_list.append(j)

arg\_list.remove(i)

return arg\_list

def check\_occurs(var, expr):

arg\_list = get\_arg\_list(expr)

if var in arg\_list:

return True

return False

def unify(expr1, expr2):

if is\_variable(expr1) and is\_variable(expr2):

if expr1 == expr2:

return 'Null'

else:

return False

elif is\_variable(expr1) and not is\_variable(expr2):

if check\_occurs(expr1, expr2):

return False

else:

tmp = str(expr2) + '/' + str(expr1)

return tmp

elif not is\_variable(expr1) and is\_variable(expr2):

if check\_occurs(expr2, expr1):

return False

else:

tmp = str(expr1) + '/' + str(expr2)

return tmp

else:

predicate\_symbol\_1, arg\_list\_1 = process\_expression(expr1)

predicate\_symbol\_2, arg\_list\_2 = process\_expression(expr2)

if predicate\_symbol\_1 != predicate\_symbol\_2:

return False

elif len(arg\_list\_1) != len(arg\_list\_2):

return False

else:

sub\_list = list()

for i in range(len(arg\_list\_1)):

tmp = unify(arg\_list\_1[i], arg\_list\_2[i])

if not tmp:

return False

elif tmp == 'Null':

pass

else:

if type(tmp) == list:

for j in tmp:

sub\_list.append(j)

else:

sub\_list.append(tmp)

return sub\_list

if \_\_name\_\_ == '\_\_main\_\_':

f1 = 'p(b(A),X,f(g(Z)))'

f2 = 'p(Z,f(Y),f(Y))'

result = unify(f1, f2)

if not result:

print('Unification failed!')

else: print('Unification successfully!')

print(result)

**PROGRAM [8]:**

P = 'P'

Q = 'Q'

R = 'R'

kb = [

(P, "=>", Q),

(Q, "=>", R),

(P,),

]

def is\_true(sentence, model):

if sentence[0] == 'not':

return not is\_true(sentence[1], model)

elif sentence[0] in model:

return model[sentence[0]]

elif len(sentence) == 1:

return False

elif sentence[1] == 'and':

return is\_true(sentence[0], model) and is\_true(sentence[2], model)

elif sentence[1] == 'or':

return is\_true(sentence[0], model) or is\_true(sentence[2], model)

elif sentence[1] == '=>':

return not is\_true(sentence[0], model) or is\_true(sentence[2], model)

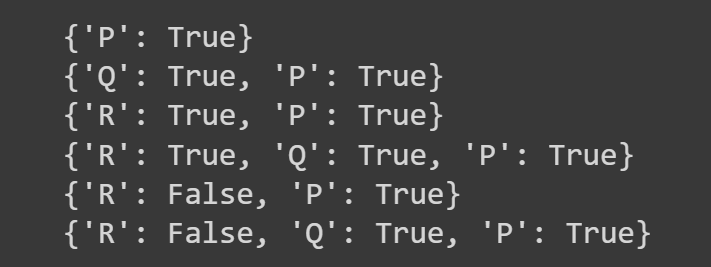
elif sentence[1] == '<=>':

return is\_true(sentence[0], model) == is\_true(sentence[2], model)

def is\_model\_satisfies\_kb(model, kb):

for sentence in kb:

**OUTPUT [8]:**



if not is\_true(sentence, model):

return False

return True

def generate\_models(symbols):

if not symbols:

return [{}]

else:

symbol = symbols[0]

rest = symbols[1:]

models = []

for model in generate\_models(rest):

models.append(model)

models.append({\*\*model, \*\*{symbol: True}})

models.append({\*\*model, \*\*{symbol: False}})

return models

symbols = [P, Q, R]

models = generate\_models(symbols)

for model in models:

if is\_model\_satisfies\_kb(model, kb):

print(model)

**PROGRAM [9]:**

import numpy as np

import matplotlib.pyplot as plt

# Generate x values for the real function

x\_func = np.linspace(-4, 4, 100)

# Corresponding y values for the real function (a simple linear relationship)

y\_func = x\_func

# Generate training data with added noise

x\_train = np.concatenate([np.random.uniform(-3, -2, 50), np.random.uniform(2, 3, 50)])

y\_train = np.concatenate([x\_train[:50] + np.random.randn(50) \* 0.5,

x\_train[50:] + np.random.randn(50) \* 0.1])

# Generate x values for the test data

x\_test = np.linspace(-10, 10, 100)

# Plotting

fig, ax = plt.subplots(1, 1, figsize=(10, 5))

# Scatter plot for training data

ax.scatter(x\_train, y\_train, label='Training Data')

# Plot the real function as a dashed line

ax.plot(x\_func, y\_func, ls='--', label='Real Function', color='green')

# Set labels and legend

ax.set\_xlabel('x')

ax.set\_ylabel('y')

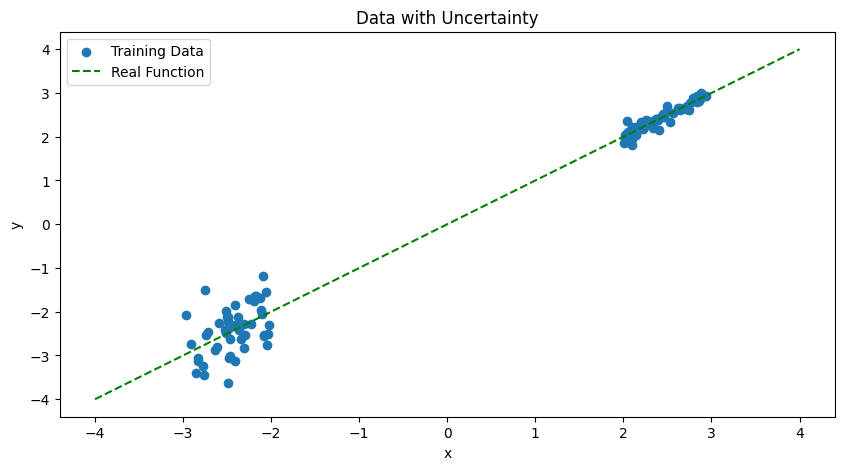
ax.legend()

# Set the title of the plot

ax.set\_title('Data with Uncertainty')

# Display the plot

plt.show()

**OUTPUT [9]:**

**PROGRAM [10]:**

class BlockWorldAgent:

def solve(self, initial, goal):

# Replace this with your actual implementation of the solve method

# This is just a placeholder

return f"Solving {initial} to {goal}"

def test():

test\_agent = BlockWorldAgent()

initial\_arrangement\_1 = [["A", "B", "C"], ["D", "E"]]

goal\_arrangement\_1 = [["A", "C"], ["D", "E", "B"]]

goal\_arrangement\_2 = [["A", "B", "C", "D", "E"]]

goal\_arrangement\_3 = [["D", "E", "A", "B", "C"]]

goal\_arrangement\_4 = [["C", "D"], ["E", "A", "B"]]

print(test\_agent.solve(initial\_arrangement\_1, goal\_arrangement\_1))

print(test\_agent.solve(initial\_arrangement\_1, goal\_arrangement\_2))

print(test\_agent.solve(initial\_arrangement\_1, goal\_arrangement\_3))

print(test\_agent.solve(initial\_arrangement\_1, goal\_arrangement\_4))

initial\_arrangement\_2 = [["A", "B", "C"], ["D", "E", "F"], ["G", "H", "I"]]

goal\_arrangement\_5 = [["A", "B", "C", "D", "E", "F", "G", "H", "I"]]

goal\_arrangement\_6 = [["I", "H", "G", "F", "E", "D", "C", "B", "A"]]

goal\_arrangement\_7 = [["H", "E", "F", "A", "C"], ["B", "D"], ["G", "I"]]

goal\_arrangement\_8 = [["F", "D", "C", "I", "G", "A"], ["B", "E", "H"]]

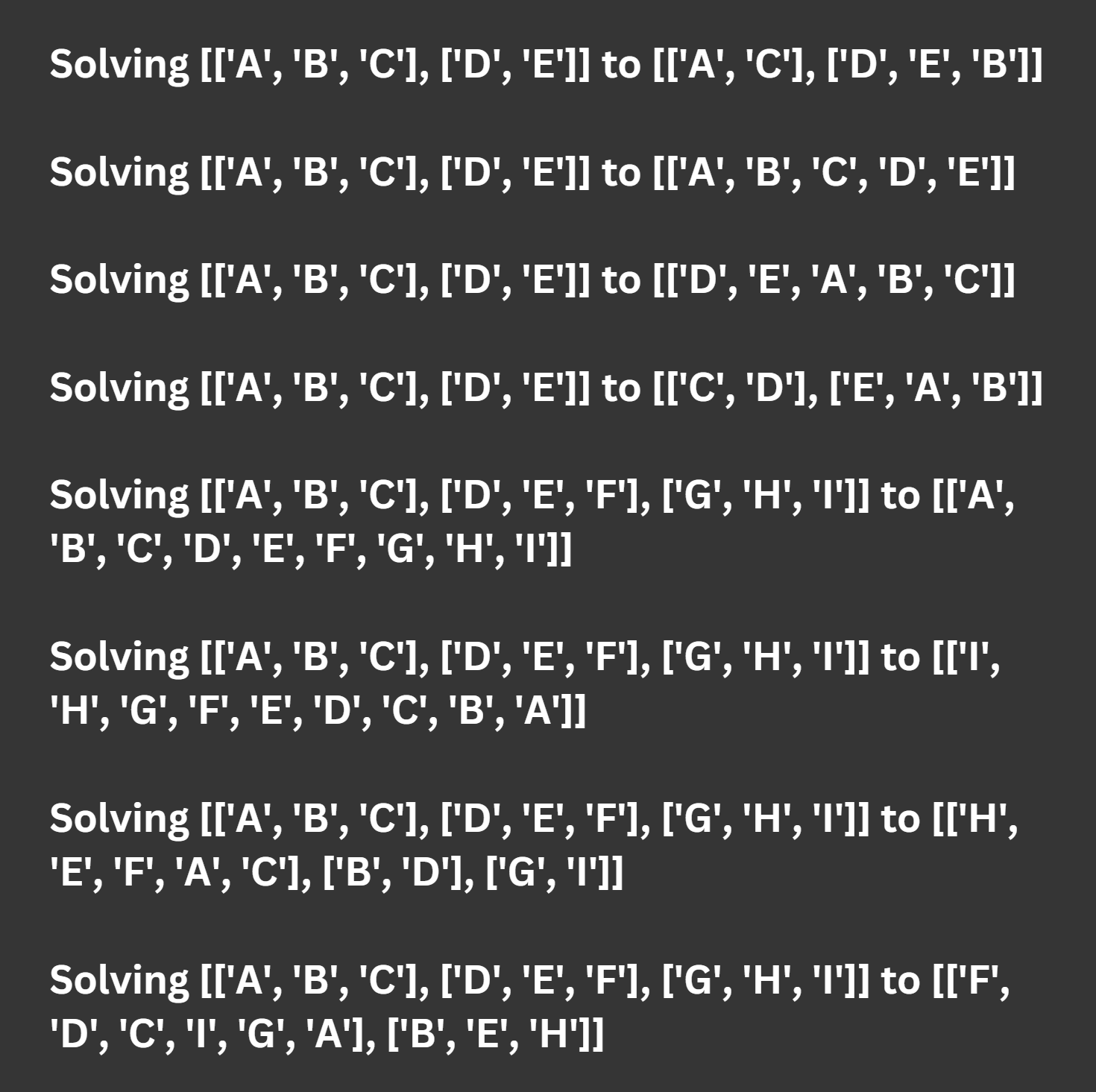
print(test\_agent.solve(initial\_arrangement\_2, goal\_arrangement\_5))

print(test\_agent.solve(initial\_arrangement\_2, goal\_arrangement\_6))

print(test\_agent.solve(initial\_arrangement\_2, goal\_arrangement\_7))

print(test\_agent.solve(initial\_arrangement\_2, goal\_arrangement\_8))

**OUTPUT [10]:**



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

test()

PROGRAM [11]:

from sklearn import datasets

from sklearn.model\_selection import train\_test\_split

from sklearn.svm import SVC

from sklearn.metrics import accuracy\_score

# Load dataset

iris = datasets.load\_iris()

X = iris.data

y = iris.target

# Split dataset into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Create SVM model

svm = SVC(kernel='linear')

# Train SVM model

svm.fit(X\_train, y\_train)

# Predict using SVM model

y\_pred = svm.predict(X\_test)

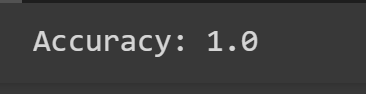
# Calculate accuracy score

accuracy = accuracy\_score(y\_test, y\_pred)

# Print accuracy score

print('Accuracy:', accuracy)

**OUTPUT [11]:**



**PROGRAM [12]:**

from sklearn.datasets import load\_breast\_cancer

from sklearn.ensemble import RandomForestClassifier, VotingClassifier

from sklearn.linear\_model import LogisticRegression

from sklearn.svm import SVC

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score

# Load dataset

data = load\_breast\_cancer()

X = data.data

y = data.target

# Split dataset into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Create individual models

lr = LogisticRegression(random\_state=42)

svc = SVC(kernel='linear', probability=True, random\_state=42)

rf = RandomForestClassifier(n\_estimators=10, random\_state=42)

# Create ensemble model

ensemble = VotingClassifier(estimators=[('lr', lr), ('svc', svc), ('rf', rf)], voting='soft')

# Train ensemble model

ensemble.fit(X\_train, y\_train)

# Predict using ensemble model

y\_pred = ensemble.predict(X\_test)

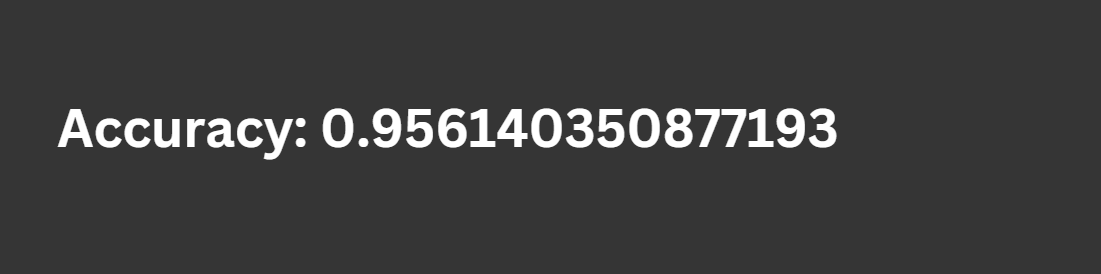
# Calculate accuracy score

accuracy = accuracy\_score(y\_test, y\_pred)

# Print accuracy score

print('Accuracy:', accuracy)

**OUTPUT [12]:**



**PROGRAM [13]:**

import nltk

nltk.download('punkt')

nltk.download('stopwords')

nltk.download('wordnet')

from nltk.corpus import stopwords

from nltk.tokenize import word\_tokenize

from nltk.stem import WordNetLemmatizer

# Define sample text

text = "Natural Language Processing (NLP) is a subfield of linguistics, computer science, and artificial intelligence concerned with the interactions between computers and human (natural) languages."

# Tokenize text

tokens = word\_tokenize(text)

# Remove stop words

stop\_words = set(stopwords.words('english'))

filtered\_tokens = [word for word in tokens if word.lower() not in stop\_words]

# Lemmatize tokens

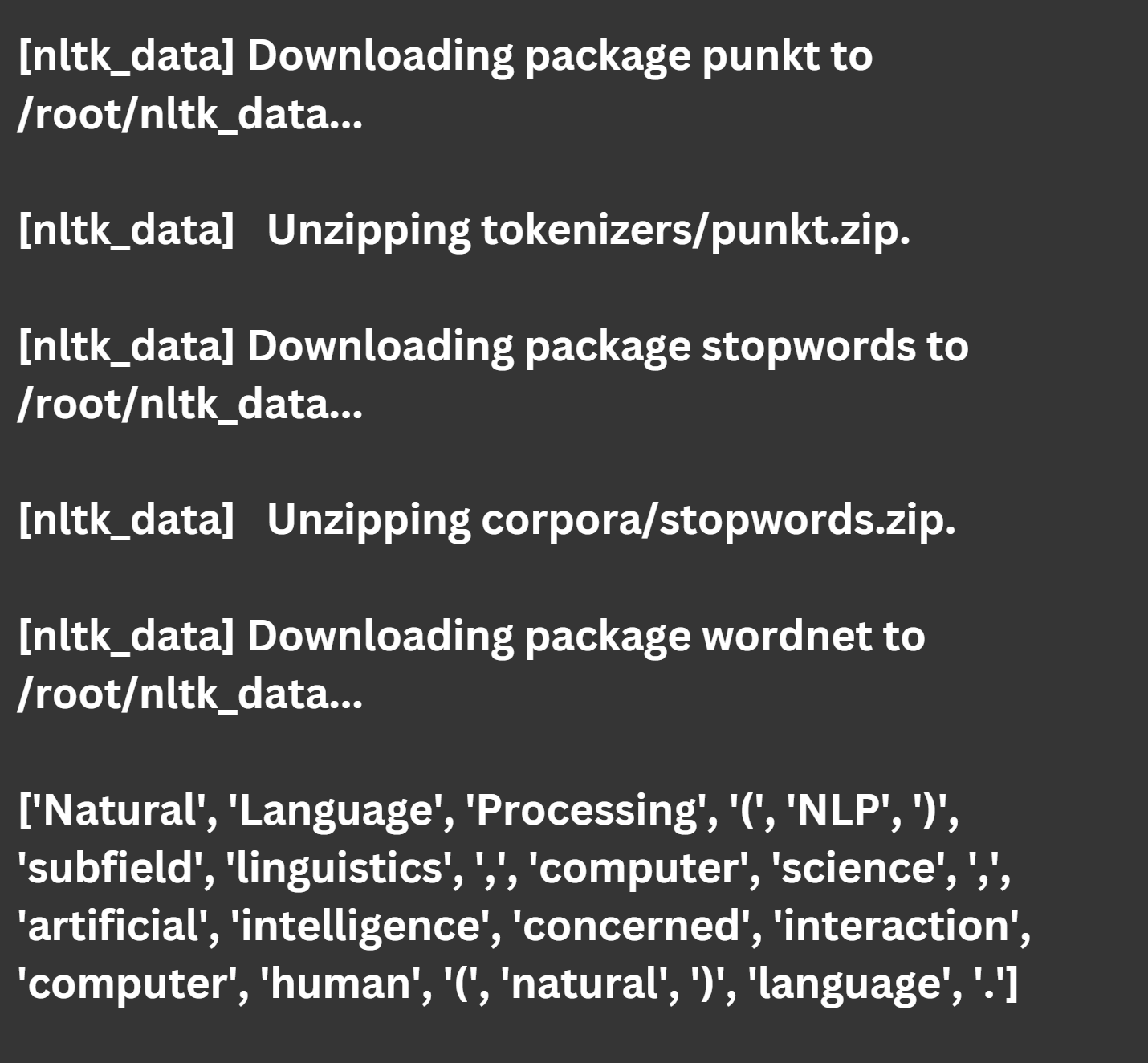
lemmatizer = WordNetLemmatizer()

lemmatized\_tokens = [lemmatizer.lemmatize(token) for token in filtered\_tokens]

# Print lemmatized tokens

print(lemmatized\_tokens)

**OUTPUT [13]:**

****

**PROGRAM [14]:**

import nltk

from nltk.tokenize import word\_tokenize

from nltk import pos\_tag

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

# Define sample text

text = """

Natural Language Processing (NLP) is a subfield of linguistics, computer science,

and artificial intelligence concerned with the interactions between computers and human

(natural) languages.

"""

# Tokenize text into words

words = word\_tokenize(text)

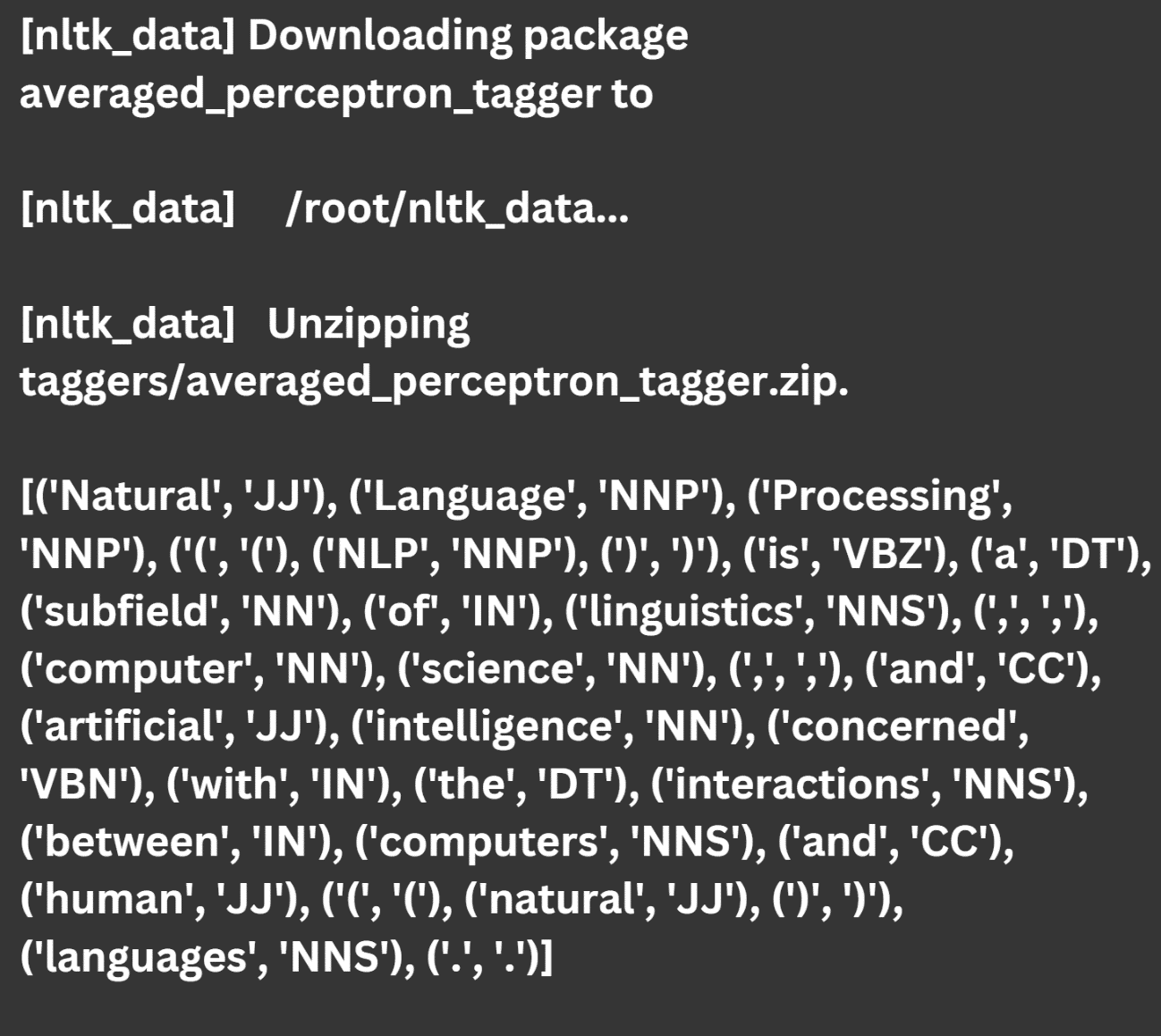
# Tag parts of speech for each word

pos\_tags = pos\_tag(words)

# Print parts of speech tags

print(pos\_tags)

**OUTPUT [14]:**



**PROGRAM [15]:**

# Importing the required libraries

from tensorflow.keras.datasets import mnist

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Conv2D, MaxPool2D, Flatten, Dense

from tensorflow.keras.layers import Dropout

# Loading data

(X\_train, y\_train), (X\_test, y\_test) = mnist.load\_data()

# Reshaping data

X\_train = X\_train.reshape((X\_train.shape[0], X\_train.shape[1], X\_train.shape[2], 1))

X\_test = X\_test.reshape((X\_test.shape[0], X\_test.shape[1], X\_test.shape[2], 1))

# Checking the shape after reshaping

print(X\_train.shape)

print(X\_test.shape)

# Normalizing the pixel values

X\_train = X\_train / 255

X\_test = X\_test / 255

# Defining model

model = Sequential()

# Adding convolution layer

model.add(Conv2D(32, (3, 3), activation='relu', input\_shape=(28, 28, 1)))

# Adding pooling layer

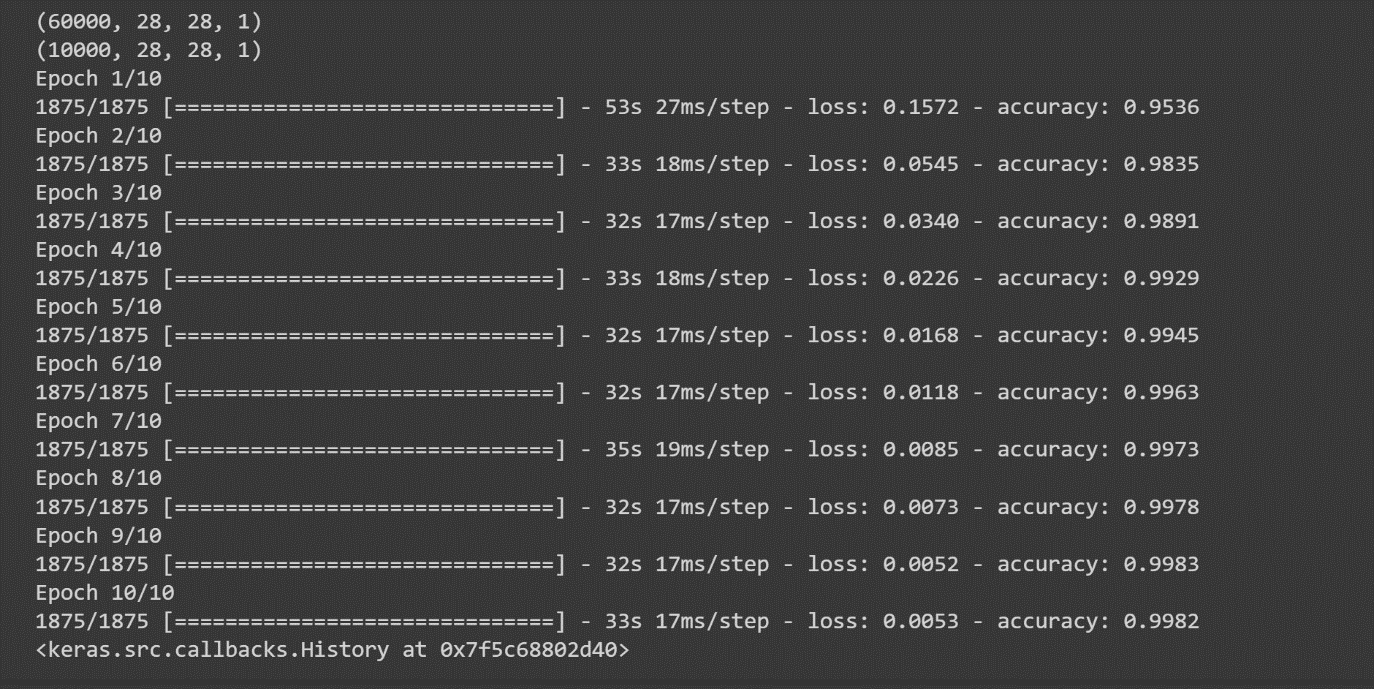
model.add(MaxPool2D(2, 2))

# Adding fully connected layer

model.add(Flatten())

model.add(Dense(100, activation='relu'))

**OUTPUT [15]:**



Adding output layer

model.add(Dense(10, activation='softmax'))

# Compiling the model

model.compile(loss='sparse\_categorical\_crossentropy', optimizer='adam', metrics=['accuracy'])

# Fitting the model

model.fit(X\_train, y\_train, epochs=10)