class MapColoring:

def \_\_init\_\_(self, variables, domains, neighbors):

self.variables = variables # Regions to be colored

self.domains = domains # Available colors for each region

self.neighbors = neighbors # Adjacent regions dictionary

self.assignment = {} # Current color assignments

def is\_consistent(self, variable, color):

"""Check if color assignment is consistent with neighbors"""

for neighbor in self.neighbors[variable]:

if neighbor in self.assignment and self.assignment[neighbor] == color:

return False

return True

def backtracking\_search(self):

"""Main backtracking algorithm"""

if len(self.assignment) == len(self.variables):

return self.assignment

# Select unassigned variable

unassigned = [v for v in self.variables if v not in self.assignment][0]

# Try each color in the domain

for color in self.domains[unassigned]:

if self.is\_consistent(unassigned, color):

self.assignment[unassigned] = color

result = self.backtracking\_search()

if result:

return result

self.assignment.pop(unassigned, None)

return None

def main():

# Example: Coloring regions of Australia

variables = ['WA', 'NT', 'SA', 'Q', 'NSW', 'V', 'T']

# Available colors for each region

colors = ['red', 'green', 'blue']

domains = {region: colors[:] for region in variables}

# Define neighboring regions

neighbors = {

'WA': ['NT', 'SA'],

'NT': ['WA', 'SA', 'Q'],

'SA': ['WA', 'NT', 'Q', 'NSW', 'V'],

'Q': ['NT', 'SA', 'NSW'],

'NSW': ['Q', 'SA', 'V'],

'V': ['SA', 'NSW'],

'T': [] # Tasmania has no neighbors

}

# Create MapColoring instance

problem = MapColoring(variables, domains, neighbors)

# Solve the problem

solution = problem.backtracking\_search()

if solution:

print("Solution found:")

for region, color in solution.items():

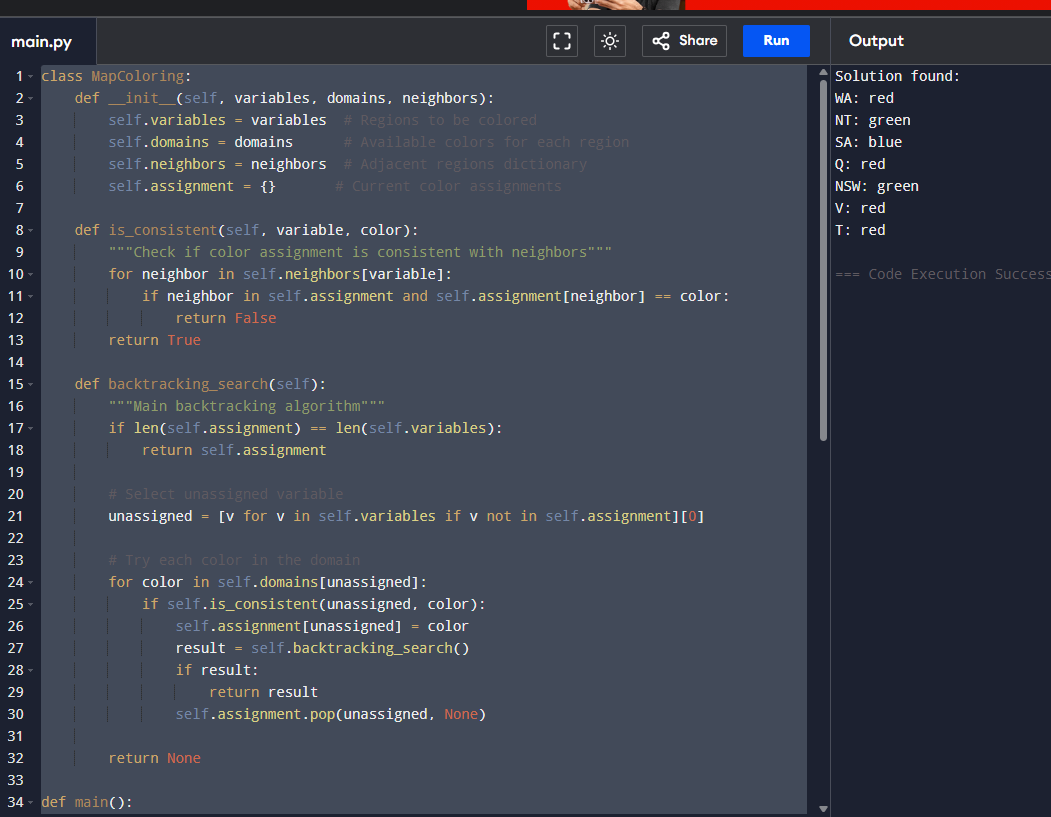
print(f"{region}: {color}")

else:

print("No solution exists!")

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

main()

def print\_board(board)

print("\n")

for row in board:

print(" | ".join(row))

print("-" \* 9)

def check\_winner(board, player):

# Check rows

for row in board:

if all(cell == player for cell in row):

return True

# Check columns

for col in range(3):

if all(board[row][col] == player for row in range(3)):

return True

# Check diagonals

if all(board[i][i] == player for i in range(3)):

return True

if all(board[i][2 - i] == player for i in range(3)):

return True

return False

def is\_full(board):

return all(cell != " " for row in board for cell in row)

def get\_move(player):

while True:

try:

move = input(f"Player {player}, enter your move (row and column: 1 1): ")

row, col = map(int, move.strip().split())

if row in [1,2,3] and col in [1,2,3]:

return row - 1, col - 1

else:

print("Invalid input. Use row and column numbers between 1 and 3.")

except ValueError:

print("Invalid format. Use two numbers separated by space.")

def play\_game():

board = [[" " for \_ in range(3)] for \_ in range(3)]

current\_player = "X"

print("Welcome to Tic Tac Toe!")

print\_board(board)

while True:

row, col = get\_move(current\_player)

if board[row][col] != " ":

print("That spot is already taken. Try again.")

continue

board[row][col] = current\_player

print\_board(board)

if check\_winner(board, current\_player):

print(f"🎉 Player {current\_player} wins!")

break

if is\_full(board):

print("It's a tie!")

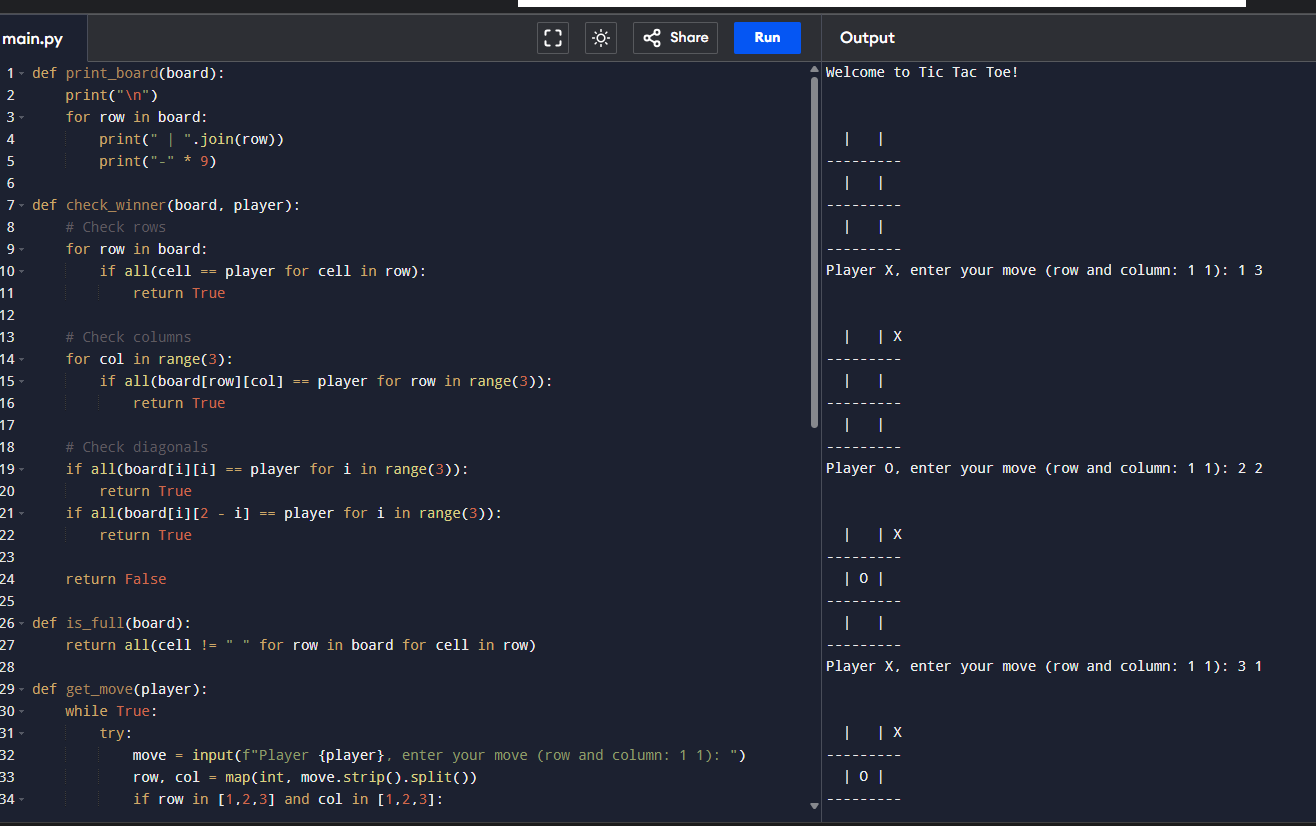
break

# Switch player

current\_player = "O" if current\_player == "X" else "X"

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

play\_game()



**Minmax algorithm for gaming**

**def winner(b, p): return any(b[i] == b[j] == b[k] == p for i, j, k in [(0,1,2),(3,4,5),(6,7,8),(0,3,6),(1,4,7),(2,5,8),(0,4,8),(2,4,6)])**

**def moves(b): return [i for i in range(9) if b[i] == ' ']**

**def minimax(b, turn):**

**if winner(b, 'X'): return 1**

**if winner(b, 'O'): return -1**

**if ' ' not in b: return 0**

**scores = []**

**for i in mo**

**b[i] = turn**

**score = minimax(b, 'O' if turn == 'X' else 'X')**

**b[i] = ' '**

**scores.append(score)**

**return max(scores) if turn == 'X' else min(scores)**

**def best\_move(b):**

**best = -2**

**move = None**

**for i in moves(b):**

**b[i] = 'X'**

**score = minimax(b, 'O')**

**b[i] = ' '**

**if score > best:**

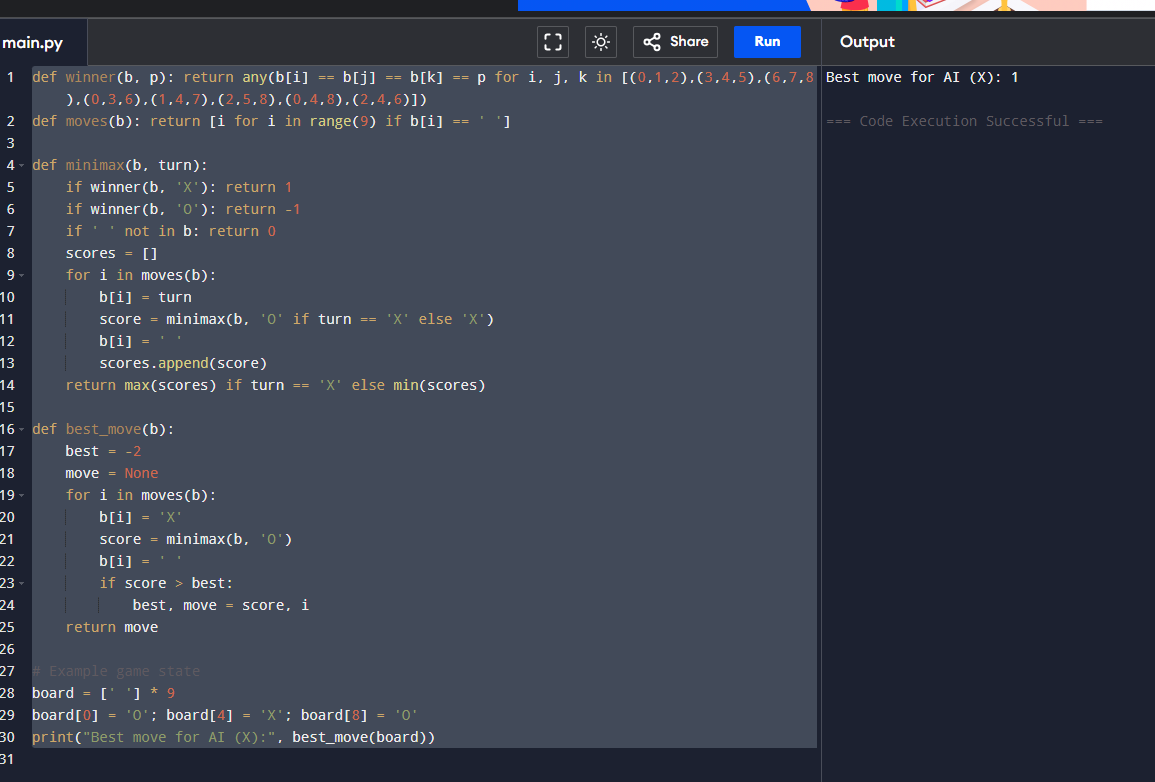
**best, move = score, i**

**return move**

**board = [' '] \* 9**

**board[0] = 'O'; board[4] = 'X'; board[8] = 'O'**

**print("Best move for AI (X):", best\_move(board))**

****

**Write the python program to implement Apha & Beta pruning algorithm for gaming**

**import math**

**def alpha\_beta\_pruning(node\_index, depth, is\_maximizing\_player, values, alpha, beta):**

**# Base case: if it's a leaf node**

**if depth == 0 or node\_index >= len(values):**

**return values[node\_index]**

**if is\_maximizing\_player:**

**max\_eval = -math.inf**

**for i in range(2): # each node has two children**

**eval = alpha\_beta\_pruning(2 \* node\_index + i, depth - 1, False, values, alpha, beta)**

**max\_eval = max(max\_eval, eval)**

**alpha = max(alpha, eval)**

**if beta <= alpha:**

**break # Beta cutoff**

**return max\_eval**

**else:**

**min\_eval = math.inf**

**for i in range(2): # each node has two children**

**eval = alpha\_beta\_pruning(2 \* node\_index + i, depth - 1, True, values, alpha, beta)**

**min\_eval = min(min\_eval, eval)**

**beta = min(beta, eval)**

**if beta <= alpha:**

**break # Alpha cutoff**

**return min\_eval**

**# Example leaf nodes (values of game outcomes at the leaves of the game tree)**

**values = [3, 5, 6, 9, 1, 2, 0, -1]**

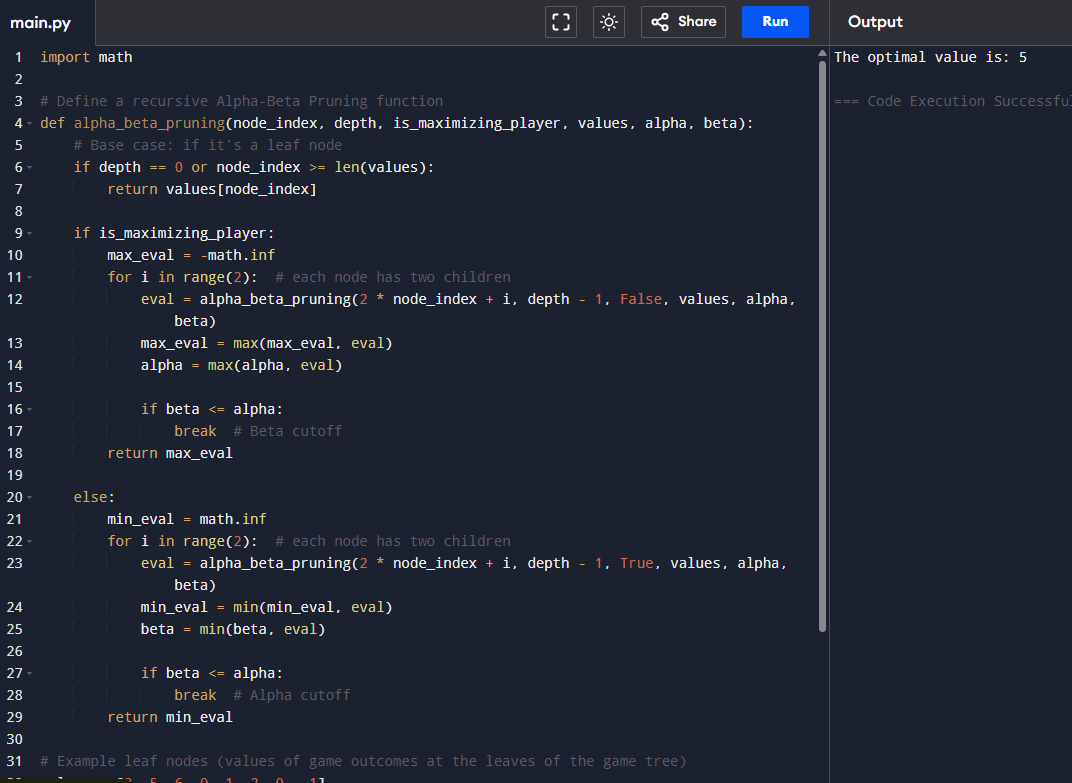
**# Tree depth (log2 of number of leaf nodes if binary tree)**

**depth = 3**

**# Run the Alpha-Beta pruning algorithm**

**result = alpha\_beta\_pruning(0, depth, True, values, -math.inf, math.inf)**

**print("The optimal value is:", result)**

****

**Write the python program to implement Decision Tree.**

**from sklearn.datasets import load\_iris**

**from sklearn.tree import DecisionTreeClassifier**

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

**from sklearn.metrics import accuracy\_score**

**iris = load\_iris()**

**X = iris.data**

**y = iris.target**

**X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)**

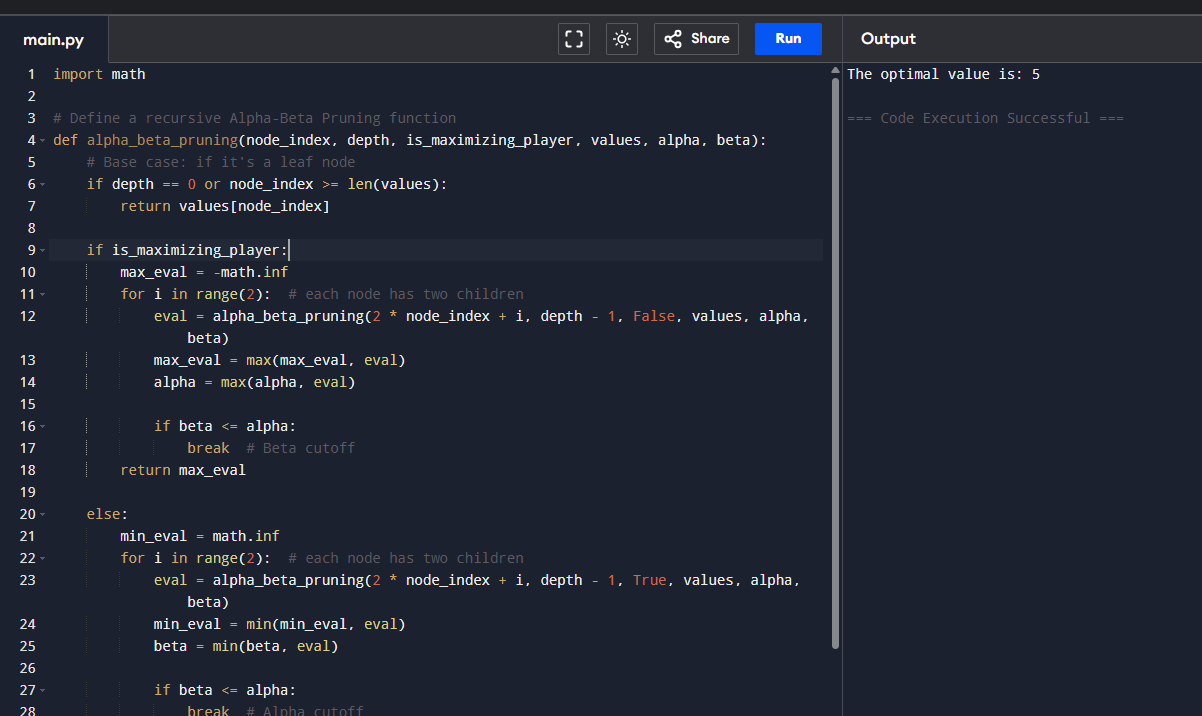
**clf = DecisionTreeClassifier()**

**clf.fit(X\_train, y\_train)**

**y\_pred = clf.predict(X\_test)**

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

**print("Accuracy:", accuracy)**

****

**Write the python program to implement Feed forward neural Network**

**import tensorflow as tf**

**from tensorflow.keras.datasets import mnist**

**from tensorflow.keras.models import Sequential**

**from tensorflow.keras.layers import Dense, Flatten**

**from tensorflow.keras.utils import to\_categorical**

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

**x\_train = x\_train / 255.0**

**x\_test = x\_test / 255.0**

**y\_train = to\_categorical(y\_train, 10)**

**y\_test = to\_categorical(y\_test, 10)**

**model = Sequential()**

**model.add(Flatten(input\_shape=(28, 28)))**

**model.add(Dense(128, activation='relu'))**

**model.add(Dense(64, activation='relu'))**

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

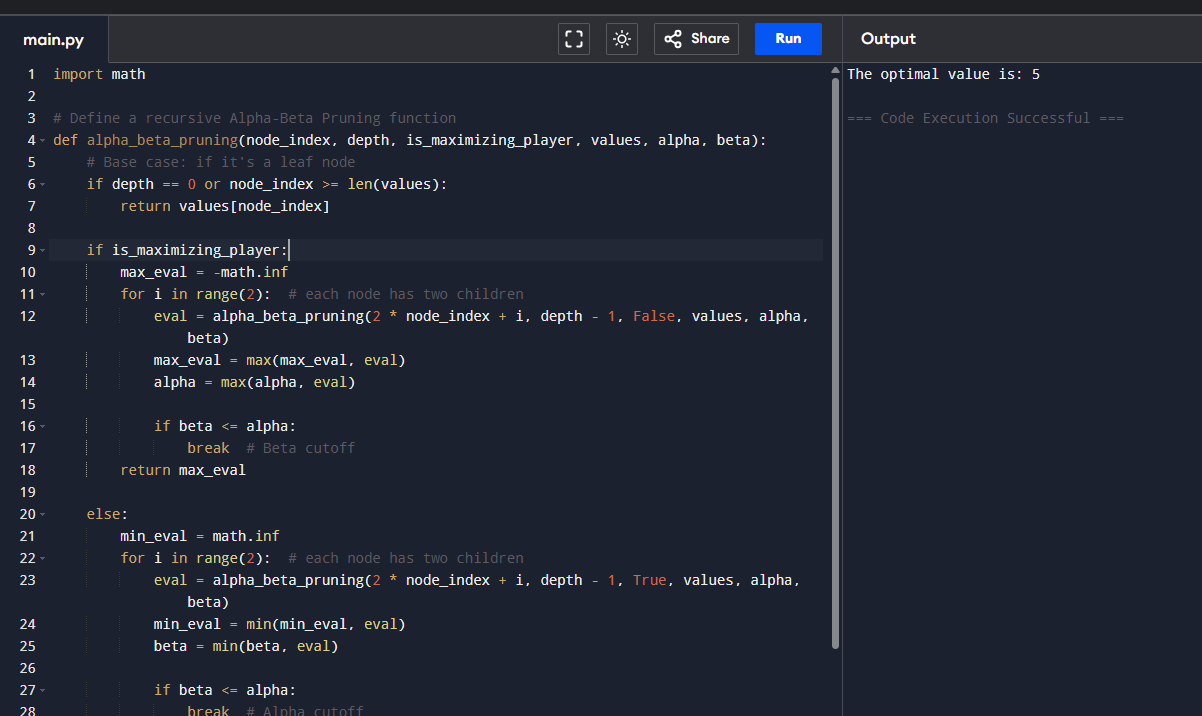
**model.compile(optimizer='adam', loss='categorical\_crossentropy', metrics=['accuracy'])**

**# Train the model**

**model.fit(x\_train, y\_train, epochs=5, batch\_size=32, validation\_split=0.1)**

**loss, accuracy = model.evaluate(x\_test, y\_test)**

**print("Test Accuracy:", accuracy)**

****