**1.Assignment One(Simple Currency converter project converted to Python)**

# Define conversion rates relative to USD

conversion\_rates = {

'USD': 1.0,

'EUR': 0.85,

'GBP': 0.73,

'JPY': 109.88,

'AUD': 1.35

}

def convert\_currency(amount, from\_currency, to\_currency):

if from\_currency not in conversion\_rates or to\_currency not in conversion\_rates:

return "Invalid currency"

# Convert the amount from the base currency to the target currency

converted\_amount = amount \* (conversion\_rates[to\_currency] / conversion\_rates[from\_currency])

return converted\_amount

def main():

print("Welcome to Simple Currency Converter")

print("Supported currencies:", ", ".join(conversion\_rates.keys()))

amount = float(input("Enter amount in USD: "))

from\_currency = 'USD'

print("Select target currency:")

for index, currency in enumerate(conversion\_rates.keys(), start=1):

print(f"Press {index} for {currency}")

target\_choice = int(input("Enter your choice: "))

target\_currencies = list(conversion\_rates.keys())

to\_currency = target\_currencies[target\_choice - 1]

converted\_amount = convert\_currency(amount, from\_currency, to\_currency)

print(f"{amount:.2f} {from\_currency} is equivalent to {converted\_amount:.2f} {to\_currency}")

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

main()

**2.Assignment Two(Depth first search solves finding connected nodes in AI)**

graph = {

0: [1, 2],

1: [3],

2: [4],

3: [],

4: []

}

def dfs(graph, node, visited, component):

# component is the list of nodes in the current connected component.

if node in visited:

return

visited.add(node)

component.append(node)

for neighbor in graph[node]:

dfs(graph, neighbor, visited, component)

def find\_connected\_components(graph):

visited = set()

connected\_components = []

for node in graph:

if node not in visited:

component = []

dfs(graph, node, visited, component)

connected\_components.append(component)

return connected\_components

connected\_components = find\_connected\_components(graph)

print(connected\_components)

**3.Assignment Three (Depth limited search )**

graph = {

'A' : ['B','C'],

'B' : ['D', 'E'],

'C' : ['F'],

'D' : [],

'E' : ['F'],

'F' : []

}

visited = set() # Set to keep track of visited nodes.

def dfs(visited, graph, node, depth\_limit):

if node not in visited and depth\_limit > 0:

print(node)

visited.add(node)

for neighbour in graph[node]:

dfs(visited, graph, neighbour, depth\_limit - 1)

# Driver Code

dfs(visited, graph, 'A', 2)

**4.Assignment Four (Hybrid search )**

graph = {

'A' : ['B','C'],

'B' : ['D', 'E'],

'C' : ['F'],

'D' : [],

'E' : ['F'],

'F' : []

}

visited = [] # to keep track of visited nodes.

queue = [] #Initialize a queue

# ////////////////////DFS

def dfs(visited, graph, node):

if node not in visited:

print (node)

print("as;la;l")

visited.append(node)

for neighbour in graph[node]:

dfs(visited, graph, neighbour)

# ////////////////////Hybrid Search///////////////////

def hybrid\_search(visited, graph, node, limit):

level = 0

visited.append(node)

queue.append(node)

# Do BFS for the first of the tree unitl the half of the given limit.

limit = 0.5 \* limit

while queue and level < limit:

s = queue.pop(0)

level += 1

print(s, end=" ")

for neighbour in graph[s]:

if neighbour not in visited:

visited.append(neighbour)

queue.append(neighbour)

# Perform DFS for the second half of the tree.

dfs(visited, graph, node)

hybrid\_search(visited ,graph ,'A' , 6)

# hybrid\_search(visited ,graph ,'A' , 3)